



e500xxxx
8/2002
Rev. 3.0

e500 Core Complex Implementation Definition

***Preliminary Draft—Subject to Change Without Notice
Motorola Confidential Proprietary***

HOW TO REACH US:**USA/EUROPE/LOCATIONS NOT LISTED:**

Motorola Literature Distribution
P.O. Box 5405, Denver, Colorado 80217
1-303-675-2140 or 1-800-441-2447

JAPAN:

Motorola Japan Ltd.
SPS, Technical Information Center
3-20-1, Minami-Azabu Minato-ku
Tokyo 106-8573 Japan
81-3-3440-3569

ASIA/PACIFIC:

Motorola Semiconductors H.K. Ltd.
Silicon Harbour Centre, 2 Dai King Street
Tai Po Industrial Estate, Tai Po, N.T., Hong Kong
852-26668334

TECHNICAL INFORMATION CENTER:

1-800-521-6274

HOME PAGE:

<http://www.motorola.com/semiconductors>

DOCUMENT COMMENTS:

FAX (512) 933-2625
Attn: RISC Applications Engineering

Information in this document is provided solely to enable system and software implementers to use Motorola products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits or integrated circuits based on the information in this document.

Motorola reserves the right to make changes without further notice to any products herein. Motorola makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Motorola assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. "Typical" parameters which may be provided in Motorola data sheets and/or specifications can and do vary in different applications and actual performance may vary over time. All operating parameters, including "Typicals" must be validated for each customer application by customer's technical experts. Motorola does not convey any license under its patent rights nor the rights of others. Motorola products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which the failure of the Motorola product could create a situation where personal injury or death may occur. Should Buyer purchase or use Motorola products for any such unintended or unauthorized application, Buyer shall indemnify and hold Motorola and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Motorola was negligent regarding the design or manufacture of the part.

**MOTOROLA**

Motorola and the Stylized M Logo are registered in the U.S. Patent and Trademark Office. digital dna is a trademark of Motorola, Inc. All other product or service names are the property of their respective owners. Motorola, Inc. is an Equal Opportunity/Affirmative Action Employer.

© Motorola, Inc. 2002

Contents

Paragraph Number	Title	Page Number
About This Book		
	Audience	xxii
	Organization.....	xxii
	Suggested Reading.....	xxiii
	General Information.....	xxiii
	Related Documentation	xxiv
	Conventions	xxv
	Terminology Conventions.....	xxvi

Part I e500 Core

Chapter 1 e500 Core Complex Overview

1.1	Overview	1-1
1.2	Features	1-5
1.3	Instruction Set	1-11
1.4	Instruction Flow	1-13
1.4.1	Branch Detection and Prediction	1-13
1.4.2	e500 Execution Pipeline	1-14
1.5	Programming Model	1-17
1.6	On-Chip Cache Implementation	1-19
1.7	Interrupts and Exception Handling	1-19
1.7.1	Exception Handling	1-19
1.7.2	Interrupt Classes	1-20
1.7.3	Interrupt Types	1-20
1.7.4	Upper Bound on Interrupt Latencies	1-21
1.7.5	Interrupt Registers.....	1-22
1.8	Memory Management.....	1-24
1.8.1	Address Translation	1-26
1.8.2	MMU Assist Registers (MAS1–MAS4 and MAS6).....	1-26
1.8.3	Process ID Registers (PID0–PID2).....	1-27
1.8.4	TLB Coherency.....	1-27
1.9	Memory Coherency	1-27

Contents

Paragraph Number	Title	Page Number
1.9.1	Atomic Update Memory References	1-28
1.9.2	Memory Access Ordering	1-28
1.9.3	Cache Control Instructions	1-28
1.9.4	Programmable Page Characteristics	1-28
1.10	Core Complex Bus (CCB)	1-29
1.11	Performance Monitoring	1-29
1.11.1	Global Control Register	1-29
1.11.2	Performance Monitor Counter Registers	1-30
1.11.3	Local Control Registers	1-30
1.12	Legacy Support of PowerPC Architecture	1-30
1.12.1	Instruction Set Compatibility	1-31
1.12.1.1	User Instruction Set	1-31
1.12.1.2	Supervisor Instruction Set	1-31
1.12.2	Memory Subsystem	1-31
1.12.3	Exception Handling	1-31
1.12.4	Memory Management	1-32
1.12.5	Reset	1-32
1.12.6	Other Differences	1-32

Chapter 2 Register Model

2.1	e500 Register Model	2-1
2.2	Register Model for 32-Bit Book E Implementations	2-3
2.2.1	Special-Purpose Registers (SPRs)	2-4
2.3	Registers for Integer Operations	2-9
2.3.1	General Purpose Registers (GPRs)	2-9
2.3.2	Integer Exception Register (XER)	2-9
2.4	Registers for Branch Operations	2-10
2.4.1	Condition Register (CR)	2-10
2.4.1.1	CR Setting for Integer Instructions	2-12
2.4.1.2	CR Setting for Store Conditional Instructions	2-12
2.4.1.3	CR Setting for Compare Instructions	2-13
2.4.2	Link Register (LR)	2-13
2.4.3	Count Register (CTR)	2-14
2.5	Processor Control Registers	2-14
2.5.1	Machine State Register (MSR)	2-14
2.5.2	Processor ID Register (PIR)	2-16
2.5.3	Processor Version Register (PVR)	2-17
2.6	Timer Registers	2-17
2.6.1	Timer Control Register (TCR)	2-19
2.6.2	Timer Status Register (TSR)	2-20

Contents

Paragraph Number	Title	Page Number
2.6.3	Time Base	2-21
2.6.4	Decrementer Register.....	2-22
2.6.5	Decrementer Auto-Reload Register (DECAR).....	2-22
2.7	Interrupt Registers.....	2-23
2.7.1	Interrupt Registers Defined by Book E.....	2-23
2.7.1.1	Save/Restore Register 0 (SRR0).....	2-23
2.7.1.2	Save/Restore Register 1 (SRR1).....	2-23
2.7.1.3	Critical Save/Restore Register 0 (CSRR0)	2-24
2.7.1.4	Critical Save/Restore Register 1 (CSRR1)	2-24
2.7.1.5	Data Exception Address Register (DEAR).....	2-25
2.7.1.6	Interrupt Vector Prefix Register (IVPR).....	2-25
2.7.1.7	Interrupt Vector Offset Registers (IVORs).....	2-25
2.7.1.8	Exception Syndrome Register (ESR)	2-26
2.7.2	e500-Specific Interrupt Registers.....	2-28
2.7.2.1	Machine Check Save/Restore Register 0 (MCSRR0)	2-28
2.7.2.2	Machine Check Save/Restore Register 1 (MCSRR1)	2-28
2.7.2.3	Machine Check Address Register (MCAR).....	2-29
2.7.2.4	Machine Check Syndrome Register (MCSR).....	2-29
2.8	Software-Use SPRs (SPRG0–SPRG7 and USPRG0).....	2-31
2.9	Branch Target Buffer (BTB) Registers	2-32
2.9.1	Branch Buffer Entry Address Register (BBEAR)	2-32
2.9.2	Branch Buffer Target Address Register (BBTAR).....	2-33
2.9.3	Branch Unit Control and Status Register (BUCSR).....	2-33
2.10	Hardware Implementation-Dependent Registers	2-34
2.10.1	Hardware Implementation-Dependent Register (HID0).....	2-35
2.10.2	Hardware Implementation-Dependent Register 1 (HID1).....	2-36
2.11	L1 Cache Configuration Registers	2-37
2.11.1	L1 Cache Control and Status Register 0 (L1CSR0)	2-37
2.11.2	L1 Cache Control and Status Register 1 (L1CSR1)	2-38
2.11.3	L1 Cache Configuration Register 0 (L1CFG0).....	2-39
2.11.4	L1 Cache Configuration Register 1 (L1CFG1).....	2-40
2.12	MMU Registers.....	2-41
2.12.1	Process ID Registers (PID0–PID2).....	2-41
2.12.2	MMU Control and Status Register 0 (MMUCSR0)	2-41
2.12.3	MMU Configuration Register (MMUCFG)	2-42
2.12.4	TLB Configuration Registers (TLBnCFG)	2-43
2.12.4.1	TLB0 Configuration Register (TLB0CFG)	2-43
2.12.4.2	TLB1 Configuration Register 1 (TLB1CFG)	2-44
2.12.5	MMU Assist Registers (MAS0–MAS4, MAS6)	2-45
2.12.5.1	MAS Register 0 (MAS0)	2-45
2.12.5.2	MAS Register 1 (MAS1)	2-46

Contents

Paragraph Number	Title	Page Number
2.12.5.3	MAS Register 2 (MAS2)	2-46
2.12.5.4	MAS Register 3 (MAS3)	2-48
2.12.5.5	MAS Register 4 (MAS4)	2-48
2.12.5.6	MAS Register 6 (MAS6)	2-49
2.13	Debug Registers	2-50
2.13.1	Debug Control Registers	2-50
2.13.1.1	Debug Control Register 0 (DBCR0)	2-50
2.13.1.2	Debug Control Register 1	2-52
2.13.1.3	Debug Control Register 2 (DBCR2)	2-53
2.13.2	Debug Status Register (DSR)	2-54
2.13.3	Instruction Address Compare Registers (IAC1–IAC4)	2-56
2.13.4	Data Address Compare Registers (DAC1–DAC2)	2-56
2.14	Signal Processing Engine (SPE) APU Registers	2-57
2.14.1	Signal Processing and Embedded Floating-Point Status and Control Register (SPEFSCR)	2-57
2.14.2	Accumulator (ACC)	2-59
2.15	Performance Monitor Registers (PMRs)	2-60
2.15.1	Global Control Register 0 (PMGC0)	2-61
2.15.2	User Global Control Register 0 (UPMGC0)	2-62
2.15.3	Local Control A Registers (PMLCa0–PMLCa3)	2-62
2.15.4	User Local Control A Registers (UPMLCa0–UPMLCa3)	2-63
2.15.5	Local Control B Registers (PMLCb0–PMLCb3)	2-63
2.15.6	User Local Control B Registers (UPMLCb0–UPMLCb3)	2-64
2.15.7	Performance Monitor Counter Registers (PMC0–PMC3)	2-64
2.15.8	User Performance Monitor Counter Registers (UPMC0–UPMC3)	2-65
2.16	Instruction Address Resume Register (IARR)	2-65
2.17	Nexus Process ID Register (NPIDR)	2-66
2.18	Book E SPR Model	2-66
2.18.1	Invalid SPR References	2-66
2.18.2	Synchronization Requirements for SPRs	2-66
2.18.3	Preserved Special-Purpose Registers	2-67
2.18.4	Reserved SPRs	2-68
2.18.5	Allocated SPRs	2-68

Chapter 3 Instruction Model

3.1	Operand Conventions	3-1
3.1.1	Data Organization in Memory and Data Transfers	3-1
3.1.2	Alignment and Misaligned Accesses	3-1
3.1.3	e500 Floating-Point Implementation	3-2
3.2	Unsupported Instructions and Instruction Forms	3-2

Contents

Paragraph Number	Title	Page Number
3.3	Instruction Set Summary.....	3-6
3.3.1	Classes of Instructions	3-7
3.3.1.1	Definition of Boundedly Undefined.....	3-7
3.3.1.2	Defined Instruction Class.....	3-8
3.3.1.3	Allocated Instruction Class.....	3-9
3.3.1.4	Preserved Instruction Class.....	3-10
3.3.1.5	Reserved Instruction Class.....	3-11
3.3.2	Instruction Forms.....	3-12
3.3.2.1	Preferred Instruction Forms.....	3-12
3.3.2.2	Invalid Instruction Forms.....	3-12
3.3.3	Addressing Modes	3-13
3.3.3.1	Memory Addressing.....	3-13
3.3.3.2	Memory Operands	3-13
3.3.3.3	Effective Address Calculation.....	3-13
3.3.3.3.1	Data Memory Addressing Modes	3-14
3.3.3.3.2	Instruction Memory Addressing Modes	3-14
3.3.3.4	Byte Ordering	3-15
3.3.3.5	Synchronization Requirements.....	3-16
3.3.3.5.1	Synchronization with tlbwe and tlbivax Instructions	3-19
3.3.3.5.2	Context Synchronization	3-19
3.3.3.5.3	Execution Synchronization.....	3-20
3.3.3.5.4	Instruction-Related Interrupts.....	3-20
3.4	Instruction Set Overview	3-21
3.4.1	Book E User-Level Instructions.....	3-22
3.4.1.1	Integer Instructions	3-22
3.4.1.1.1	Integer Arithmetic Instructions	3-22
3.4.1.1.2	Integer Compare Instructions	3-23
3.4.1.1.3	Integer Logical Instructions	3-24
3.4.1.1.4	Integer Rotate and Shift Instructions	3-24
3.4.1.2	Load and Store Instructions	3-25
3.4.1.2.1	Self-Modifying Code	3-26
3.4.1.2.2	Integer Load and Store Address Generation	3-26
3.4.1.2.3	Integer Load Instructions.....	3-29
3.4.1.2.4	Integer Store Instructions.....	3-30
3.4.1.2.5	Integer Load and Store with Byte-Reverse Instructions	3-31
3.4.1.2.6	Integer Load and Store Multiple Instructions	3-31
3.4.1.3	Branch and Flow Control Instructions.....	3-32
3.4.1.3.1	Branch Instruction Address Calculation	3-32
3.4.1.3.2	Branch Relative Addressing Mode	3-32
3.4.1.3.3	Branch Conditional to Relative Addressing Mode	3-33
3.4.1.3.4	Branch to Absolute Addressing Mode.....	3-34

Contents

Paragraph Number	Title	Page Number
3.4.1.3.5	Branch Conditional to Absolute Addressing Mode	3-35
3.4.1.3.6	Branch Conditional to Link Register Addressing Mode.....	3-35
3.4.1.3.7	Branch Conditional to Count Register Addressing Mode	3-36
3.4.1.4	Conditional Branch Control.....	3-37
3.4.1.4.1	Branch Instructions.....	3-39
3.4.1.4.2	Condition Register Logical Instructions	3-40
3.4.1.4.3	Trap Instructions	3-40
3.4.1.5	System Linkage Instruction	3-40
3.4.1.6	Processor Control Instructions.....	3-41
3.4.1.6.1	Move to/from Condition Register Instructions	3-41
3.4.1.6.2	Move To/From Special-Purpose Register Instructions	3-41
3.4.1.7	Memory Synchronization Instructions	3-45
3.4.1.8	Atomic Update Primitives Using lwarx and stwcx.....	3-46
3.4.1.8.1	Reservations.....	3-48
3.4.1.8.2	Forward Progress	3-50
3.4.1.8.3	Reservation Loss Due to Granularity	3-50
3.4.1.9	Memory Control Instructions	3-51
3.4.1.9.1	User-Level Cache Instructions.....	3-51
3.4.2	Supervisor-Level Instructions	3-53
3.4.2.1	System Linkage Instructions.....	3-54
3.4.2.2	Supervisor-Level Memory Control Instructions	3-54
3.4.2.2.1	Supervisor-Level Cache Instruction	3-55
3.4.2.2.2	Supervisor-Level TLB Management Instructions	3-55
3.4.3	Recommended Simplified Mnemonics	3-56
3.5	Book E Instructions with Implementation-Specific Features	3-57
3.5.1	New e500 Instructions	3-57
3.5.2	Context Synchronization.....	3-58
3.6	Instruction Fetching	3-58
3.7	Branch Prediction.....	3-58
3.8	Memory Access Alignment Support	3-58
3.9	Update Instructions	3-59
3.10	Memory Synchronization.....	3-60
3.11	e500-Specific Instructions.....	3-60
3.11.1	Signal Processing Engine (SPE) APU Instructions	3-61
3.11.1.1	SPE APU Vector Instructions.....	3-61
3.11.1.2	SPE APU Floating-Point Instructions.....	3-68
3.11.2	Branch Target Buffer (BTB) Locking Instructions	3-70
3.11.3	Instruction Select (isel) APU	3-70
3.11.4	Performance Monitor APU	3-71
3.11.5	Cache Locking APU	3-71

Contents

Paragraph Number	Title	Page Number
3.11.6	Machine Check APU	3-73
3.12	Instruction Listing.....	3-73

Chapter 4 Execution Timing

4.1	Terminology and Conventions	4-1
4.2	Instruction Timing Overview	4-4
4.3	General Timing Considerations	4-11
4.3.1	General Instruction Flow	4-12
4.3.2	Instruction Fetch Timing Considerations.....	4-13
4.3.2.1	L1 and L2 TLB Access Times	4-13
4.3.2.2	Interrupts Associated with Instruction Fetching	4-14
4.3.2.3	Cache-Related Latency	4-14
4.3.3	Dispatch, Issue, and Completion Considerations	4-15
4.3.3.1	GPR and CR Rename Register Operation	4-16
4.3.3.2	LR and CTR Shadow (Speculative) Registers	4-17
4.3.3.3	Instruction Serialization	4-18
4.3.4	Interrupt Latency.....	4-19
4.3.5	Memory Synchronization Instruction Timing Considerations.....	4-19
4.4	Execution Unit Timings	4-20
4.4.1	Branch Unit Execution Timing	4-20
4.4.1.1	Branch Instructions and Completion	4-20
4.4.1.2	Branch Prediction and Resolution	4-21
4.4.1.3	Branch Table Buffer (BTB)	4-22
4.4.1.3.1	BTB Locking APU Programming Model.....	4-25
4.4.1.3.2	BTB Operations	4-26
4.4.1.3.3	BTB Special Cases—Phantom Branches and Multiple Matches	4-26
4.4.2	Load/Store Unit Queueing Structures.....	4-27
4.4.2.1	Load/Store Unit (LSU)	4-29
4.4.3	Simple and Multiple Unit Execution Timings	4-29
4.4.4	SPE Floating-Point Execution Timing.....	4-29
4.4.5	Load/Store Unit Execution Timing.....	4-30
4.4.5.1	Effect of Operand Placement on Performance	4-30
4.5	Memory Performance Considerations	4-31
4.6	Instruction Latency Summary.....	4-31
4.7	Instruction Scheduling Guidelines.....	4-43
4.7.1	Fetch/Branch Considerations	4-44
4.7.1.1	Dynamic Prediction versus No Branch Prediction	4-44
4.7.1.1.1	Position-Independent Code Example	4-44
4.7.2	Dispatch Unit Resource Requirements	4-44
4.7.2.1	Dispatch Groupings	4-45

Contents

Paragraph Number	Title	Page Number
4.7.3	Issue Queue Resource Requirements.....	4-45
4.7.3.1	General Issue Queue (GIQ)	4-45
4.7.4	Completion Unit Resource Requirements	4-45
4.7.4.1	Completion Groupings.....	4-46
4.7.5	Serialization Effects	4-46
4.7.6	Execution Unit Considerations	4-46
4.7.6.1	SU Considerations	4-46
4.7.6.2	MU Considerations	4-47
4.7.6.2.1	Load/Store Interaction	4-47
4.7.6.2.2	Misalignment Effects	4-47
4.7.6.2.3	Load Miss Pipeline	4-48

Chapter 5 Interrupts and Exceptions

5.1	Overview	5-1
5.2	e500 Interrupt Definition.....	5-2
5.2.1	Recoverability from Interrupts.....	5-3
5.3	Interrupt Registers.....	5-3
5.4	Exceptions.....	5-7
5.5	Interrupt Classes.....	5-8
5.5.1	Requirements for System Reset Generation	5-10
5.6	Interrupt Processing	5-11
5.7	Interrupt Definitions.....	5-12
5.7.1	Critical Input Interrupt.....	5-14
5.7.2	Machine Check Interrupt	5-15
5.7.3	Data Storage Interrupt.....	5-17
5.7.4	Instruction Storage Interrupt.....	5-19
5.7.5	External Input Interrupt	5-20
5.7.6	Alignment Interrupt	5-21
5.7.7	Program Interrupt.....	5-22
5.7.8	System Call Interrupt.....	5-24
5.7.9	Decrementer Interrupt.....	5-24
5.7.10	Fixed-Interval Timer Interrupt	5-25
5.7.11	Watchdog Timer Interrupt.....	5-26
5.7.12	Data TLB Error Interrupt	5-27
5.7.13	Instruction TLB Error Interrupt	5-29
5.7.14	Debug Interrupt.....	5-29
5.7.15	e500-Specific Exceptions.....	5-30
5.7.15.1	SPE APU Unavailable Exception	5-30
5.7.15.2	SPE Floating-Point Data Exception.....	5-31
5.7.15.3	SPE Floating-Point Round Exception	5-31

Contents

Paragraph Number	Title	Page Number
5.8	Performance Monitor Interrupt	5-32
5.9	Partially Executed Instructions	5-32
5.10	Interrupt Ordering and Masking	5-34
5.10.1	Guidelines for System Software	5-35
5.10.2	Interrupt Order	5-35
5.11	Exception Priorities.....	5-36
5.11.1	e500 Interrupt Priorities	5-37
5.11.2	Exception Priorities for Integer Load, Store, and Cache Management Instructions	5-38
5.11.3	Exception Priorities for Allocated Load and Store Instructions	5-38
5.11.4	Exception Priorities for Allocated Instructions (Other)	5-39
5.11.5	Exception Priorities for Privileged Instructions	5-39
5.11.6	Exception Priorities for Trap Instructions.....	5-40
5.11.7	Exception Priorities for System Call Instruction	5-40
5.11.8	Exception Priorities for Branch Instructions	5-40
5.11.9	Exception Priorities for Return from Interrupt Instructions	5-41
5.11.10	Exception Priorities for SPE Instructions	5-41
5.11.11	Exception Priorities for Reserved Instructions	5-42
5.12	Upper Bound on Interrupt Latencies.....	5-42

Chapter 6 Power Management

6.1	Overview	6-1
6.2	Power Management States	6-1
6.2.1	Full On	6-2
6.2.2	Halted	6-2
6.2.3	Power Down.....	6-2
6.2.3.1	Disabling the Timer	6-3
6.2.3.2	Further Power Reduction	6-3
6.3	Interrupts and Power Management	6-3
6.4	Snooping in the Power Down State	6-4
6.5	Power Management Signals.....	6-4
6.6	Power Management Control Bits.....	6-5
6.7	Software Considerations for Power Management	6-5

Chapter 7 Performance Monitor

7.1	Overview	7-1
7.2	Performance Monitor APU Programming Model.....	7-2
7.2.1	Performance Monitor APU Registers	7-2

Contents

Paragraph Number	Title	Page Number
7.2.1.1	Global Control Register 0 (PMGC0)	7-3
7.2.1.2	User Global Control Register 0 (UPMGC0).....	7-4
7.2.1.3	Local Control A Registers (PMLCa0–PMLCa3).....	7-4
7.2.1.4	User Local Control A Registers (UPMLCa0–UPMLCa3)	7-5
7.2.1.5	Local Control B Registers (PMLCb0–PMLCb3)	7-6
7.2.1.6	User Local Control B Registers (UPMLCb0–UPMLCb3).....	7-6
7.2.1.7	Performance Monitor Counter Registers (PMC0–PMC3).....	7-7
7.2.1.8	User Performance Monitor Counter Registers (UPMC0–UPMC3)	7-7
7.3	Performance Monitor APU Instructions	7-8
7.4	Performance Monitor Interrupt	7-8
7.5	Event Counting	7-8
7.5.1	Processor Context Configurability	7-8
7.6	Example Usage	7-9
7.6.1	Chaining Counters	7-9
7.6.2	Thresholding	7-10
7.7	Event Selection	7-10

Chapter 8 Debug Support

8.1	Overview	8-1
8.1.1	Deviations from Book E Debug Mode	8-1
8.1.2	Software Facilities	8-2
8.1.3	Hardware Facilities	8-2
8.2	TAP Controller and Register Model	8-2
8.2.1	TAP Interface Signals	8-2
8.2.1.1	JTAG Test Clock (TCK)—Input.....	8-3
8.2.1.2	JTAG Test Data Input (TDI)—Input.....	8-3
8.2.1.3	JTAG Test Data Output (TDO)—Output.....	8-4
8.2.1.4	JTAG Test Mode Select (TMS)—Input	8-4
8.2.1.5	JTAG Test Reset ($\overline{\text{TRST}}$)—Input.....	8-4
8.2.1.6	TAP Enable (TAP_EN)—Input	8-4
8.2.1.7	Test Data Out Enable (TDO_EN)—Output.....	8-4
8.2.1.8	TLM Selected (TLMSEL)—Output	8-4
8.3	Unconditional Debug Event.....	8-5
8.4	Book E Debug Events	8-5
8.4.1	Instruction Address Compare Debug Event.....	8-6
8.4.1.1	Instruction Address Compare User/Supervisor Mode	8-6
8.4.1.2	EffectiveAddress Mode.....	8-6
8.4.1.3	Instruction Address Compare Mode	8-7
8.4.2	Data Address Compare Debug Event	8-8
8.4.2.1	Data Address Compare Read/Write Enable.....	8-8

Contents

Paragraph Number	Title	Page Number
8.4.2.2	Data Address Compare User/Supervisor Mode	8-9
8.4.2.3	Effective Address Mode	8-9
8.4.2.4	Data Address Compare (DAC) Mode	8-9
8.4.3	Trap Debug Event	8-10
8.4.4	Branch Taken (BRT) Debug Event	8-11
8.4.5	Instruction Complete Debug Event.....	8-11
8.4.6	Interrupt Taken (IRPT) Debug Event.....	8-12
8.4.7	Return Debug Event.....	8-12
8.4.8	Unconditional Debug Event.....	8-13

Part II e500 Core Complex

Chapter 9 Timer Facilities

9.1	Timer Facilities	9-1
9.2	Timer Registers	9-2
9.2.1	Hardware Implementation-Dependent Register 0	9-2
9.2.2	Timer Control Register (TCR).....	9-2
9.2.3	Timer Status Register (TSR).....	9-4
9.2.4	Time Base	9-4
9.2.4.1	Writing the TB	9-7
9.2.4.2	Reading the TB	9-7
9.2.4.3	Computing Time of Day from the Time Base	9-7
9.2.4.3.1	Non-Constant Update Frequency	9-9
9.3	The e500 Timer Implementation.....	9-9
9.3.1	Decrementer Register.....	9-10
9.3.2	Decrementer Auto-Reload Register (DECAR).....	9-10
9.3.3	Decrementer Operation.....	9-10
9.4	Fixed-Interval Timer	9-12
9.5	Watchdog Timer.....	9-13
9.6	Freezing the Timer Facilities	9-15

Chapter 10 Auxiliary Processing Units (APUs)

10.1	Instruction Select APU.....	10-1
10.1.1	Instruction Select APU Programming Model	10-2
10.1.2	Using isel to Improve Conditional Branch Performance.....	10-2
10.2	Performance Monitor APU	10-2

Contents

Paragraph Number	Title	Page Number
10.3	Signal Processing Engine APU (SPE APU).....	10-4
10.3.1	SPE APU Programming Model	10-4
10.3.1.1	SPE APU Register Model	10-4
10.3.1.1.1	Allocated Bits in Book E–Defined Registers.....	10-5
10.3.1.1.2	Signal Processing and Embedded Floating-Point Status and Control Register (SPEFSCR)	10-5
10.3.1.1.3	Accumulator	10-6
10.3.1.1.4	SPE Interrupt Vector Offset Registers	10-8
10.3.1.2	SPE APU Instruction Model	10-8
10.3.1.2.1	PowerPC UISA and Book E Instructions	10-8
10.3.1.2.2	IEEE 754 Compliance	10-9
10.3.1.2.3	Signal-Processing Floating-Point (SPFPU) Operations	10-9
10.3.2	SPE APU Exceptions	10-9
10.3.2.1	SPE APU Unavailable Exception	10-10
10.3.2.2	Vector Floating-Point Data Exception	10-10
10.3.2.3	Vector Floating-Point Round Exception	10-10
10.3.2.4	Vector Alignment Exception.....	10-11
10.3.2.5	Exception Priorities	10-11
10.3.3	SPE APU Functions	10-11
10.4	Branch Target Buffer (BTB) Locking APU	10-13
10.4.1	BTB Locking APU Programming Model	10-14
10.4.1.1	Branch Target Buffer Instructions.....	10-14
10.4.1.2	Branch Target Buffer Registers.....	10-14
10.5	Cache Block Lock and Unlock APU	10-14
10.5.1	Cache Block Lock and Unlock APU Programming Model	10-15
10.5.1.1	Cache Block Lock and Unlock APU Instruction Model	10-15
10.5.1.2	Cache Locking APU Register Model.....	10-16
10.5.1.3	Cache Locking APU Interrupt Model.....	10-16
10.6	Machine Check APU.....	10-16
10.6.1	Machine Check APU Programming Model	10-16
10.6.1.1	Machine Check APU Register Model.....	10-16
10.6.1.2	Machine Check APU Instruction Model.....	10-17
10.6.1.3	Machine Check Interrupt	10-17

Chapter 11 Cache and MMU Background

11.1	Memory and Cache Coherency.....	11-1
11.1.1	Memory/Cache Access Attributes (WIMGE Bits).....	11-2
11.1.1.1	Write-Through Attribute	11-3
11.1.1.2	Caching-Inhibited Attribute	11-3
11.1.1.3	Memory-Coherence-Required Attribute	11-3

Contents

Paragraph Number	Title	Page Number
11.1.1.4	Guarded Attribute	11-4
11.1.1.4.1	Definition of Speculative and Out-of-Order Memory Accesses	11-4
11.1.1.4.2	Performing Operations Speculatively	11-5
11.1.1.4.3	Guarded Memory	11-5
11.1.1.4.4	Speculative Accesses to Guarded Memory	11-6
11.1.1.4.5	Instruction Accesses: Guarded Memory and No-Execute Memory	11-6
11.1.1.5	Endianness	11-6
11.1.1.5.1	Big-Endian Pages	11-7
11.1.1.5.2	Little-Endian Pages	11-7
11.1.1.5.3	Structure Mapping Examples	11-7
11.1.1.6	Mismatched Memory Cache Attributes	11-8
11.1.1.7	Coherency Paradoxes and WIMGE	11-9
11.1.2	Cache Control Instructions	11-10
11.1.2.1	Self-Modifying Code	11-10
11.1.3	Shared Memory	11-11
11.1.3.1	Memory Access Ordering	11-11
11.1.3.1.1	Programming Considerations	11-13
11.1.3.1.2	Programming Examples	11-15
11.1.3.2	Lock Acquisition and Import Barriers	11-16
11.1.3.2.1	Acquire Lock and Import Shared Memory	11-16
11.1.3.2.2	Obtain Pointer and Import Shared Memory	11-16
11.1.3.3	Atomic Memory References	11-17
11.2	Book E MMU Architecture	11-17
11.2.1	Virtual Address	11-19
11.2.1.1	Address Spaces	11-19
11.2.1.1.1	Instruction Address Spaces	11-20
11.2.1.1.2	Data Address Spaces	11-20
11.2.1.2	Process ID	11-21
11.2.2	Book E TLB Concept	11-21
11.2.3	Address Translation	11-23
11.2.3.1	Page Size and Effective Address Bits Compared	11-25
11.2.3.2	Permission Attribute Comparison	11-26
11.2.3.3	Page Size and Real Address Generation	11-27
11.2.4	Permission Attributes	11-28
11.2.4.1	Execute Access Permission	11-28
11.2.4.2	Read Access Permission	11-29
11.2.4.3	Write Access Permission	11-29
11.2.4.4	Permission Control and Cache Management Instructions	11-30
11.2.4.5	Permissions Control and String Instructions	11-31
11.2.4.6	Use of Permissions to Maintain Page History	11-31
11.2.5	Crossing Page Boundaries	11-31

Contents

Paragraph Number	Title	Page Number
11.3	Motorola Book E MMU Architecture.....	11-31
11.3.1	Address Translation and the Motorola Standard.....	11-32
11.3.1.1	Variable Number of PID Registers and Sharing	11-32
11.3.1.2	Match Criteria for TLB Entries	11-33
11.3.1.3	Translation Algorithms	11-34
11.3.2	TLB Entries.....	11-35
11.3.2.1	IPROT TLB Entry Field.....	11-35
11.3.3	Reading and Writing TLB Entries	11-35
11.3.3.1	Reading TLB Entries	11-36
11.3.3.2	Writing TLB Entries	11-36
11.3.4	TLB Entry Invalidations	11-36
11.3.5	MMU Assist (MAS) Registers.....	11-38
11.3.5.1	MAS Register Formats and Bit Definitions	11-39
11.3.6	MAS Registers and Exception Handling	11-44
11.3.6.1	TLB Miss Exception Types	11-45
11.3.6.1.1	Instruction TLB Error Interrupt Settings	11-45
11.3.6.1.2	Data TLB Error Interrupt Settings	11-45
11.3.6.1.3	TLB Miss Exception MAS Register Settings	11-45
11.3.6.2	Permissions Violation Exception Types.....	11-46
11.3.6.2.1	Instruction Storage Interrupt Settings.....	11-47
11.3.6.2.2	Data Storage Interrupt Settings.....	11-47
11.3.6.3	MAS Register Update Summary for Exceptions, tlbsx, and tlbre	11-47
11.3.7	TLB Configuration Information.....	11-48
11.3.7.1	MMU Configuration Register (MMUCFG)	11-49
11.3.7.2	TLB Configuration Registers (TLBnCFG).....	11-49

Chapter 12 L1 Caches

12.1	Overview	12-1
12.1.1	Block Diagram	12-2
12.1.1.1	Load/Store Unit (LSU)	12-3
12.1.1.1.1	Caching-allowed Loads and LSU	12-4
12.1.1.1.2	LSU Store Queue	12-4
12.1.1.1.3	LSU L1 Load Miss Queue (LMQ)	12-4
12.1.1.1.4	LSU Data Line Fill Buffer (DLFB)	12-4
12.1.1.1.5	LSU Data Write Buffer (DWB).....	12-5
12.1.1.2	Instruction Unit	12-5
12.1.1.3	Core Interface Unit	12-5
12.2	L1 Cache Organization	12-5
12.2.1	L1 Data Cache Organization.....	12-5
12.2.2	L1 Instruction Cache Organization.....	12-7

Contents

Paragraph Number	Title	Page Number
12.3	Cache Coherency Support.....	12-8
12.3.1	Data Cache Coherency Model	12-8
12.3.1.1	Shared Memory Multiprocessor Support.....	12-10
12.3.2	Instruction Cache Coherency Model	12-11
12.3.3	Snoop Signaling	12-11
12.3.4	WIMGE	12-12
12.3.4.1	Write-Back Stores.....	12-12
12.3.4.2	Write-Through Stores	12-13
12.3.4.3	Caching-Inhibited Loads and Stores.....	12-13
12.3.4.4	Misaligned Accesses and the Endian (E) Bit	12-13
12.3.4.5	Speculative Accesses to Guarded Memory	12-13
12.3.5	Load/Store Operations	12-14
12.3.5.1	Performed Loads and Stores	12-14
12.3.5.2	Sequential Consistency of Memory Accesses	12-14
12.3.5.3	Enforcing Store Ordering with Respect to Loads	12-15
12.3.5.4	Atomic Memory References	12-15
12.4	L1 Cache Control	12-15
12.4.1	Cache Control Instructions	12-16
12.4.2	L1 Instruction and Data Cache Enabling/Disabling	12-18
12.4.3	L1 Instruction and Data Cache Flash Invalidation.....	12-18
12.4.4	L1 Instruction and Data Cache Line Locking/Unlocking.....	12-19
12.4.4.1	Effects of Other Cache Instructions on Locked Lines.....	12-20
12.4.4.2	Flash Clearing of Lock Bits.....	12-21
12.5	L1 Data Cache Flushing.....	12-21
12.6	L1 Cache Operation	12-22
12.6.1	Cache Miss and Reload Operations	12-22
12.6.1.1	Data Cache Fills.....	12-22
12.6.1.2	Instruction Cache Fills	12-23
12.6.1.3	Cache Allocation on Misses.....	12-23
12.6.1.4	Store Miss Merging	12-23
12.6.1.5	Store Hit to a Data Cache Block Marked Shared	12-24
12.6.1.6	Data Cache Block Push Operation	12-24
12.6.2	L1 Cache Block Replacement Selection.....	12-24
12.6.2.1	PLRU Replacement	12-24
12.6.2.2	PLRU Bit Updates	12-25
12.6.2.3	Cache Locking and PLRU	12-26

Chapter 13 Memory Management Units

13.1	e500 MMU Overview	13-1
13.1.1	MMU Features	13-1

Contents

Paragraph Number	Title	Page Number
13.1.2	TLB Entry Maintenance Features.....	13-2
13.2	Effective-to-Real Address Translation	13-3
13.2.1	Virtual Addresses with Three PID Registers	13-4
13.2.2	Variable-Sized Pages.....	13-5
13.2.3	Checking for TLB Entry Hit	13-5
13.2.4	Checking for Access Permissions	13-6
13.3	Translation Lookaside Buffers (TLBs)	13-6
13.3.1	L1 TLB Arrays	13-8
13.3.2	L2 TLB Arrays	13-9
13.3.2.1	IPROT Invalidation Protection in TLB1	13-10
13.3.2.2	SHAREN Page Attribute	13-11
13.3.2.3	Replacement Algorithms for L2 MMU	13-11
13.3.3	Consistency Between L1 and L2 TLBs	13-12
13.3.4	L1 and L2 TLB Access Times	13-13
13.3.5	The G Bit (of WIMGE).....	13-14
13.3.6	TLB Entry Field Definitions	13-14
13.4	TLB Instructions—Implementation.....	13-15
13.4.1	TLB Read Entry (tlbre) Instruction	13-15
13.4.1.1	Reading Entries from the TLB1 Array	13-16
13.4.1.2	Reading Entries from the TLB0 Array	13-16
13.4.2	TLB Write Entry (tlbwe) Instruction	13-16
13.4.2.1	Writing to the TLB1 Array	13-16
13.4.2.2	Writing to the TLB0 Array	13-17
13.4.3	TLB Search (tlbsx) Instruction—Searching the TLB1 and TLB0 Arrays... 13-17	
13.4.4	TLB Invalidate (tlbivax) Instruction	13-17
13.4.4.1	TLB Selection for tlbivax Instruction.....	13-19
13.4.4.2	Invalidate All Address Encoding for tlbivax Instruction.....	13-19
13.4.4.3	TLB Invalidate Broadcast Enabling.....	13-19
13.4.5	TLB Synchronize (tlbsync) Instruction	13-19
13.5	TLB Entry Maintenance—Details	13-20
13.5.1	Automatic Updates—TLB Miss Exceptions	13-21
13.5.2	TLB Interrupt Routines.....	13-21
13.5.2.1	TLB Error Interrupt Handlers	13-22
13.5.2.2	Permissions Violations (ISI, DSI) Interrupt Handlers	13-22
13.6	TLB States after Reset	13-22
13.7	Core Complex MMU Registers	13-23
13.7.1	e500 MAS Registers	13-24
13.7.2	MAS Register Updates	13-29

Contents

Paragraph Number	Title	Page Number

Appendix A Simplified Mnemonics

A.1	Symbols	A-1
A.2	Simplified Mnemonics for Subtract Instructions	A-2
A.2.1	Subtract Immediate	A-2
A.2.2	Subtract	A-2
A.3	Simplified Mnemonics for Compare Instructions.....	A-3
A.3.1	Word Comparisons.....	A-3
A.4	Simplified Mnemonics for Rotate and Shift Instructions	A-3
A.4.1	Operations on Words.....	A-4
A.5	Simplified Mnemonics for Branch Instructions.....	A-5
A.5.1	BO and BI Fields	A-5
A.5.2	Basic Branch Mnemonics	A-6
A.5.3	Branch Mnemonics Incorporating Conditions.....	A-10
A.5.4	Branch Prediction	A-13
A.6	Simplified Mnemonics for Condition Register Logical Instructions	A-14
A.7	Simplified Mnemonics for Trap Instructions	A-15
A.8	Simplified Mnemonics for SPRs.....	A-17
A.9	Recommended Simplified Mnemonics	A-21
A.9.1	No-Op (nop)	A-21
A.9.2	Load Immediate (li)	A-22
A.9.3	Load Address (la).....	A-22
A.9.4	Move Register (mr)	A-22
A.9.5	Complement Register (not)	A-22
A.9.6	Move to Condition Register (mtcr).....	A-23
A.10	e500-Specific Simplified Mnemonics	A-23
A.10.1	Integer Select (isel)	A-23
A.10.2	SPU Mnemonics	A-23
A.11	Complete List of Simplified Mnemonics.....	A-23

Appendix B Programming Examples

B.1	Synchronization	B-1
B.1.1	Synchronization Primitives.....	B-2
B.1.1.1	Fetch and No-op	B-2
B.1.1.2	Fetch and Store	B-3
B.1.1.3	Fetch and Add	B-3
B.1.1.4	Fetch and AND	B-3
B.1.1.5	Test and Set.....	B-3
B.1.1.6	Compare and Swap	B-4

Contents

Paragraph Number	Title	Page Number
B.1.2	Lock Acquisition and Release.....	B-4
B.1.3	List Insertion.....	B-5
B.1.4	Notes	B-6
B.2	Multiple-Precision Shifts	B-7
B.2.1	Notes	B-9

Appendix C Guidelines for 32-Bit Book E

C.1	64-Bit-Specific Book E Instructions	C-1
C.2	Registers on 32-Bit Book E Implementations	C-2
C.3	Addressing on 32-Bit Book E Implementations	C-2
C.4	TLB Fields on 32-bit Book E Implementations.....	C-2
C.5	32-Bit Book E Software Guidelines	C-3
C.5.1	32-Bit Instruction Selection	C-3
C.5.2	32-Bit Addressing	C-3

About This Book

The primary objective of this user's manual is to describe the functionality of the e500 embedded microprocessor core for software and hardware developers. This book is intended as a companion to the *EREF: A Reference for Motorola Book E and the e500 Core*.

Book E is a PowerPC architecture definition for embedded processors that ensures binary compatibility with the user-instruction set architecture (UISA) portion of the PowerPC architecture as it was jointly developed by Apple, IBM, and Motorola. The version of the architecture jointly developed by Apple, IBM, and Motorola is referred to as the AIM version of the PowerPC architecture.

This book contains information about Book E and the e500, as shown in Table i.

Table i. EREF Contents

Instructions and registers defined in Book E or in an auxiliary processing unit (APU) that are implemented in the e500
Instructions and registers defined in Book E that are not implemented in the e500. These features appear as gray text.
Instructions and registers defined for the e500 that are not in Book E

This document distinguishes between the three levels of the architectural and implementation definition, as follows:

- The Book E architecture —Book E defines a set of user-level instructions and registers that are drawn from the user instruction set architecture (UISA) portion of the AIM definition PowerPC architecture. Book E also include numerous other supervisor-level registers and instructions as they were defined in the AIM version of the PowerPC architecture for the virtual environment architecture (VEA) and the operating environment architecture (OEA).

Because Book E defines a much different model for operating system resources such as the MMU and interrupts, it defines many new registers and instructions.

- Motorola Book E implementation standards. In many cases, the Book E architecture definition provides a very general framework, leaving many higher-level details up to the implementation. To ensure consistency among its Book E implementations, Motorola has defined implementation standards that provide an additional layer of architecture between Book E and the actual devices.

- e500 implementation details. Each processor typically defines instructions, registers, bits within registers, and other aspects that are more detailed than either the Book E definition or the Motorola Book E implementation standards.

This book describes all of the instructions and registers implemented on the e500, including those defined by Book E and those that are e500-specific.

Information in this book is subject to change without notice, as described in the disclaimers on the title page of this book. As with any technical documentation, it is the readers' responsibility to be sure they are using the most recent version of the documentation.

Audience

It is assumed that the reader understands operating systems, microprocessor system design, and the basic principles of RISC processing.

Organization

Following is a summary and a brief description of the major sections of this manual:

- Chapter 1, “e500 Core Complex Overview,” provides a general description of e500 functionality.
- Chapter 2, “Register Model,” is useful for software engineers who need to understand the programming model for the three programming environments and the functionality of each register.
- Chapter 3, “Instruction Model,” provides an overview of the addressing modes and a description of the instructions. Instructions are organized by function.
- Chapter 4, “Execution Timing,” describes how instructions are fetched, decoded, issues, executed, and completed and how instruction results are preset to the processor and memory system. Tables are provided that indicate latency and throughput for each of the instructions supported by the e500.
- Chapter 5, “Interrupts and Exceptions,” describes how the e500 implements the interrupt model as it is defined by the Book E architecture.
- Chapter 6, “Power Management,” describes the power management facilities as they are defined by Book E and implemented in the e500 core.
- Chapter 7, “Performance Monitor,” describes the e500 implementation of the performance monitor APU that is defined by the Motorola Book E implementation standards.
- Chapter 8, “Debug Support,” describes the debug facilities as they are defined by Book E and implemented in the e500 core.

- Chapter 9, “Timer Facilities,” describes the Book E-defined timer facilities implemented in the e500 core. These resources include the time base (TB), decremter (DEC), fixed-interval timer (FIT), and watchdog timer.
- Chapter 10, “Auxiliary Processing Units (APUs),” describes the extensions to the Book E-defined programming model that are supported on the e500. These include the following:
 - Section 10.1, “Instruction Select APU”
 - Section 10.2, “Performance Monitor APU”
 - Section 10.3, “Signal Processing Engine APU (SPE APU)”
 - Section 10.4, “Branch Target Buffer (BTB) Locking APU”
 - Section 10.5, “Cache Block Lock and Unlock APU”
 - Section 10.6, “Machine Check APU”
- Chapter 11, “Cache and MMU Background,” describes the cache and memory management models as they are defined by Book E and further specified by the Motorola Book E implementation standards (EIS). Specific details about the e500 cache and MMU implementations are provided in subsequent chapters.
- Chapter 12, “L1 Caches,” provides specific hardware and software details regarding the e500 cache implementation.
- Chapter 13, “Memory Management Units,” provides specific hardware and software details regarding the e500 MMU implementation.
- Appendix A, “Simplified Mnemonics,” provides a set of simplified mnemonic examples and symbols.
- Appendix B, “Programming Examples,” gives examples of how the memory synchronization instructions can be used to emulate various synchronization primitives and to provide more complex forms of synchronization.
- Appendix C, “Guidelines for 32-Bit Book E,” provides a set of guidelines for software developers. Application software written to these guidelines can be labelled 32-bit Book E applications and can expect to execute properly on all implementations of Book E, both 32-bit and 64-bit implementations.

Suggested Reading

This section lists additional reading that provides background for the information in this manual as well as general information about the architecture.

General Information

The following documentation, published by Morgan-Kaufmann Publishers, 340 Pine Street, Sixth Floor, San Francisco, CA, provides useful information about the PowerPC architecture and computer architecture in general:

- *The PowerPC Architecture: A Specification for a New Family of RISC Processors*, Second Edition, by International Business Machines, Inc.
For updates to the specification, see <http://www.austin.ibm.com/tech/ppc-chg.html>.
- *Computer Architecture: A Quantitative Approach*, Second Edition, by John L. Hennessy and David A. Patterson
- *Computer Organization and Design: The Hardware/Software Interface*, Second Edition, David A. Patterson and John L. Hennessy

Related Documentation

Motorola documentation is available from the sources listed on the back cover of this manual; the document order numbers are included in parentheses for ease in ordering:

- *EREF: A Reference for Motorola Book E and the e500 Core*—This book provides a higher-level view of the programming model as it is defined by Book E, the Motorola Book E implementation standards, and the e500 microprocessor.
- User's manuals—These books provide details about individual implementations.
- Addenda/errata to user's manuals—Because some processors have follow-on parts an addendum is provided that describes the additional features and functionality changes. These addenda are intended for use with the corresponding user's manuals.
- Hardware specifications—Hardware specifications provide specific data regarding bus timing, signal behavior, and AC, DC, and thermal characteristics, as well as other design considerations.
- Technical summaries—Each device has a technical summary that provides an overview of its features. This document is roughly the equivalent to the overview (Chapter 1) of an implementation's user's manual.
- Application notes—These short documents address specific design issues useful to programmers and engineers working with Motorola processors.

Additional literature is published as new processors become available. For a current list of documentation, refer to <http://www.motorola.com/semiconductors>.

Conventions

This document uses the following notational conventions:

gray text	Features defined by Book E that are not supported in the e500 are shown in gray text.
cleared/set	When a bit takes the value zero, it is said to be cleared; when it takes a value of one, it is said to be set.
mnemonics	Instruction mnemonics are shown in lowercase bold.
<i>italics</i>	Italics indicate variable command parameters, for example, bcctr <i>x</i> . Book titles in text are set in italics Internal signals are set in italics, for example, <i>qual BG</i>
0x0	Prefix to denote hexadecimal number
0b0	Prefix to denote binary number
rA, rB	Instruction syntax used to identify a source GPR
rD	Instruction syntax used to identify a destination GPR
frA, frB, frC	Instruction syntax used to identify a source FPR
frD	Instruction syntax used to identify a destination FPR
REG[FIELD]	Abbreviations for registers are shown in uppercase text. Specific bits, fields, or ranges appear in brackets. For example, MSR[LE] refers to the little-endian mode enable bit in the machine state register.
x	In some contexts, such as signal encodings, an unitalicized x indicates a don't care.
<i>x</i>	An italicized x indicates an alphanumeric variable.
<i>n</i>	An italicized n indicates a numeric variable.
¬	NOT logical operator
&	AND logical operator
	OR logical operator
0000	Indicates reserved bits or bit fields in a register. Although these bits can be written to as ones or zeros, they are always read as zeros.

Additional conventions used with instruction encodings are described in Table 3-2.

Terminology Conventions

Table ii lists certain terms used in this manual that differ from the architecture terminology conventions.

Table ii. Terminology Conventions

The Architecture Specification	This Manual
Extended mnemonics	Simplified mnemonics
Privileged mode (or privileged state)	Supervisor level
Problem mode (or problem state)	User level
Out of order memory accesses	Speculative memory accesses
Storage (locations)	Memory
Storage (the act of)	Access

Part I

e500 Core

Part I specifically describes the e500 core, excluding details about cache memories and MMU features. It contains chapters that apply to the entire core, as follows:

- Chapter 1, “e500 Core Complex Overview,” summarizes the e500 core. This a 32-bit implementation of the Book E PowerPC architecture, including a recognition that different processor implementations may require extensions or deviations from the architectural descriptions.
- Chapter 2, “Register Model,” describe the e500 core register models as defined in Book E and the additional implementation-specific registers unique to the e500 core, including a Book E SPR model.
- Chapter 3, “Instruction Model,” provides information about the Book E architecture as it relates specifically to the e500 core complex. The e500 core complex also implements several APU, which define additional instructions, registers, and interrupts. The chapter also features operand conventions, branch prediction, memory access alignment support, and memory synchronization sections.
- Chapter 4, “Execution Timing,” describes the e500 core’s operations performance as defined by instructions and how it reports the results of instruction execution. It gives detailed descriptions of how the core execution units work and how these units interact with other parts of the processor, such as the instruction fetching mechanism, register files, and caches. Included are examples of instruction sequences and tables that provide information useful to assembly language programmers for optimizing performance.
- Chapter 5, “Interrupts and Exceptions,” is a general description of the Book E interrupt and exception model and gives details of the additions and changes to that model that are implemented in the e500 core complex.
- Chapter 6, “Power Management,” describes the hardware and software resources the system uses to minimize its power consumption. This chapter regards the power management facilities as they are defined by Book E and implemented in devices that contain the e500 core, but its scope is limited to features of the core only.

- Chapter 7, “Performance Monitor,” describes the e500 implementation of the performance monitor APU that is defined by the Motorola Book E implementation standards.
 - Chapter 8, “Debug Support,” describes the e500 core complex internal debug capabilities and associated features. Included are important deviations to the Book E debug mode.
- .

Chapter 1

e500 Core Complex Overview

This chapter provides an overview of the e500 microprocessor core. It includes the following:

- An overview of the Book E version of the PowerPC architecture features as implemented in this core and a summary of the core feature set
- A summary of the instruction pipeline and flow
- An overview of the programming model
- An overview of interrupts and exception handling
- A description of the memory management architecture
- High-level details of the e500 core memory and coherency model
- A brief description of the core complex bus (CCB)
- Signal descriptions
- A summary of the Book E architecture compatibility and migration from the original version of the PowerPC architecture as it was defined by Apple, IBM, and Motorola (referred to as the AIM version of the PowerPC architecture).

1.1 Overview

The e500 processor core is a low-power implementation of the family of reduced instruction set computing (RISC) embedded processors that implement the Book E definition of the PowerPC architecture. The e500 implements the 32-bit portion of the Book E architecture using the lower words in the 64-bit general-purpose registers (GPRs).

Book E allows processors to provide auxiliary processing units (APUs), which are extensions to the architecture that can perform computational or system management functions. The most significant of these on the e500 is the signal processing engine APU (SPE APU), which includes a suite of vector instructions that use the upper and lower halves of the GPRs as a single two-element operand. Most APUs implemented on the e500 are defined by the Motorola Book E implementation standards (EIS).

The e500 provides 32-bit effective addresses, integer data types of 8, 16, and 32 bits, and a 32-bit floating-point data type. For detailed information regarding the Book E instruction set, see Chapter 3, “Instruction Model.” The SPE APU on the e500 also supports signed and unsigned fractional data types of 32 bits.

Overview

Figure 1-1 is a block diagram of the processor core complex that shows how the functional units operate independently and in parallel. Note that this conceptual diagram does not attempt to show how these features are physically implemented.

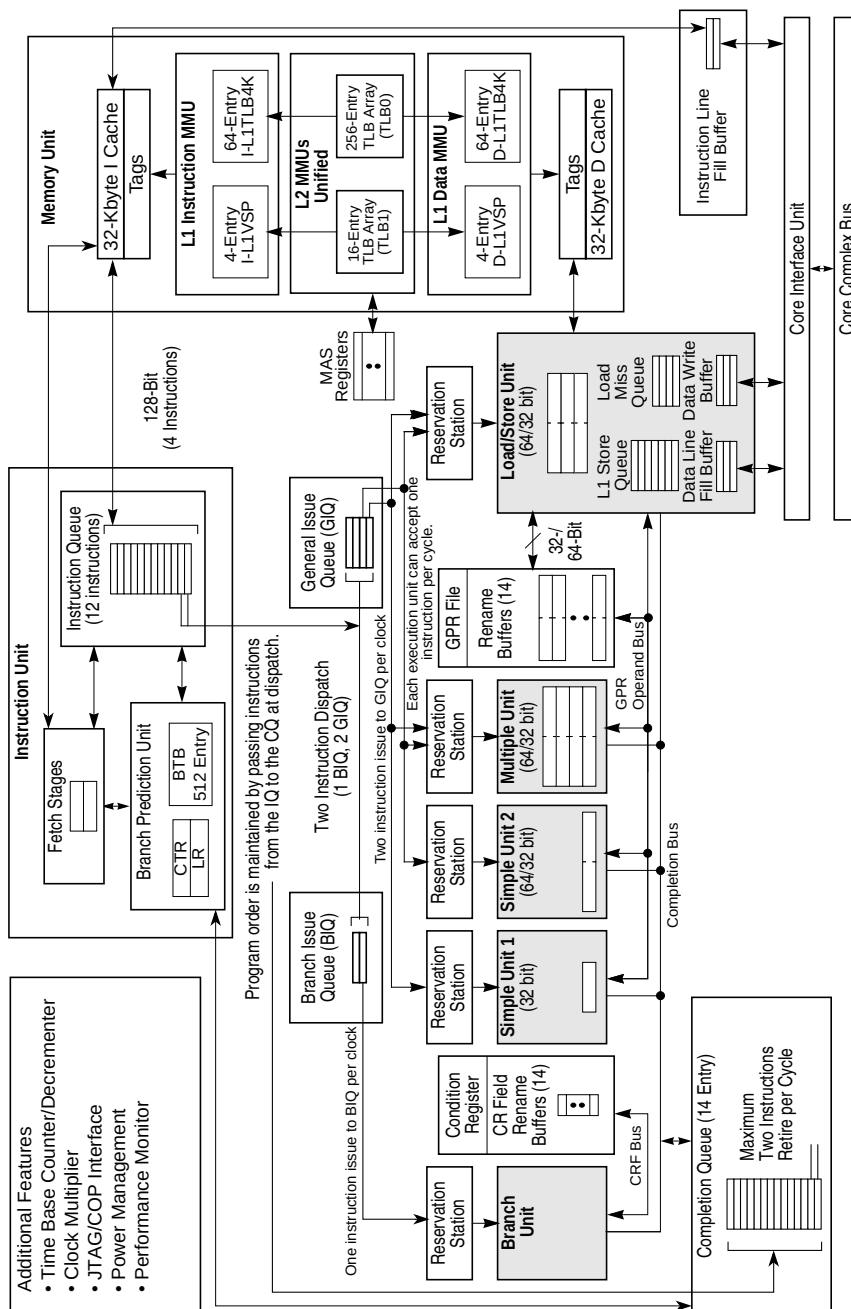


Figure 1-1. e500 Core Complex Block Diagram

The core complex is a superscalar processor that can issue two instructions and complete two instructions per clock cycle. Instructions complete in order, but can execute out of order. Execution results are available to subsequent instructions through the rename buffers, but those results are recorded into architected registers in program order, maintaining precise exception model. All arithmetic instructions that execute in the core operate on data in the GPRs. Although the GPRs are 64-bits wide, only SPE APU vector instructions operate on the upper word of the GPRs; the upper 32-bits of the GPRs are not affected by 32-bit instructions.

The processor core integrates two simple instruction units (SU1, SU2), a multiple-cycle instruction unit (MU), a branch unit (BU), and a load/store unit (LSU). The MU, LSU, and SU1 support 64-bit SIMD/signal processing instructions. The ability to execute five instructions in parallel and the use of simple instructions with short execution times yield high efficiency and throughput. Most integer instructions execute in one clock cycle. A series of independent vector floating-point add instructions can be issued and completed with a throughput of one instruction per cycle.

The core complex includes independent, on-chip, 32-Kbyte, eight-way set-associative, physically addressed caches for instructions and data. It also includes on-chip first-level instruction and data memory management units (MMUs) and an on-chip second-level unified MMU.

- The first-level MMUs contain two four-entry, fully-associative instruction and data translation lookaside buffer (TLB) arrays that provide support for demand-paged virtual memory address translation and variable-sized pages. They also contain two 64-entry, four-way set-associative, instruction and data TLB arrays that support 4-Kbyte pages. All of these arrays are maintained completely by the hardware with a least-recently-used (LRU) algorithm.
- The second-level MMU contains a 16-entry fully-associative unified (instruction and data) TLB array that provides support for variable-sized pages and a 256-entry, two-way set-associative unified TLB for 4-Kbyte page size support. These TLBs are maintained completely by the software.

The core complex allows cache-line-based user-mode locks on the contents in either the instruction or data cache. This provides embedded applications with the capability for locking interrupt routines or other important (time-sensitive) instruction sequences into the instruction cache. It also allows data to be locked into the data cache, which supports deterministic execution time.

The core complex supports a high-speed on-chip internal bus with data tagging called the core complex bus (CCB). The CCB has two general purpose read data buses, one write data bus, data parity bits, data tag bits, an address bus and address attribute bits. The processor core complex supports out-of-order reads, in-order writes, and one level of pipelining for addresses with address-retry responses. It can also support single-beat and burst data transfers for memory accesses and memory-mapped I/O operations.

1.2 Features

Key features of the e500 are summarized as follows:

- Implements Book E 32-bit architecture
- Additional auxiliary processing units.

The following APUs are defined by the Motorola Book E implementation standards (EIS).

- Instruction select. This APU consists of the Instruction Select instruction, **isel**, which is a conditional register move that helps eliminate conditional branches, decreases latency, and reduces the code footprint.
- Performance monitor. The performance monitor facility provides the ability to monitor and count predefined events such as processor clocks, misses in the instruction cache or data cache, types of instructions decoded, or mispredicted branches. The count of such events can be used to trigger the performance monitor exception. Additional performance monitor registers (PMRs) similar to SPRs are used to configure and track performance monitor operations. These registers are accessed with the Move to PMR and Move from PMR instructions (**mtpmr** and **mfpmr**). For more information, see Section 1.11, “Performance Monitoring.”
- Cache block lock and unlock. This APU allows instructions and data to be locked into their respective caches on a cache line basis. Locking is performed by a set of touch-and-lock set instructions. This functionality can be enabled for user mode by setting MSR[UCLE]. The APU also provides resources for detecting and handling overlocking conditions.
- Machine check. The machine check interrupt is treated as a separate level of interrupt. It uses its own save and restore registers (MCSRR0 and MCSRR1) and Return from Machine Check Interrupt (**rfmci**) instruction. See Section 1.7, “Interrupts and Exception Handling.”
- Signal processing engine APU (SPE APU). Note that the SPE is not a separate unit; SPE computational and logical instructions are executed in the simple and multiple-cycle units used by all other computational and logical instructions, and SPE loads and stores use executed in the common LSU. Figure 1-1 shows how execution logic for SU1, the MU, and the LSU is replicated to support operations on the upper halves of the GPRs.

To support the SPE, the e500 register set is modified in the following ways:

- GPRs are widened to 64 bits to support SPE vector operations. Note that the upper 32 bits are affected only by SPE instructions.
- A 64-bit accumulator (ACC) has been added

Features

- The signal processing and embedded floating-point status and control register (SPEFSCR) provides interrupt control and status, as well as various condition bits associated with the operations performed by the SPE APU.

These registers are shown in Figure 1-6. SPE instructions are grouped as follows:

- Single-cycle integer add and subtract with the same latencies for SPE APU operations as for the 32-bit equivalent
- Single-cycle logical operations
- Single-cycle shift and rotate
- Four-cycle integer pipelined multiplies
- 4-, 11-, 19-, and 35-cycle integer divides
- If **rA** or **rB** is zero, a floating-point divide takes 4 cycles. All other cases take 29 cycles.
- 4-cycle SIMD pipelined multiply-accumulate (MAC)
- 64-bit accumulator for no-stall MAC operations
- Single-precision floating-point single-instruction, multiple-data (SIMD) operations.
- Single-precision scalar floating-point single-instructions. These instructions are listed in Table 1-2.

The branch target buffer (BTB) locking is specific to the e500. The BTB locking APU gives the user the ability to lock, unlock, and invalidate BTB entries. Note that the BTB locking APU is an e500 cor feature; it is not part of the EIS.

- Cache structure—Separate 32-Kbyte, 32-byte line, 8-way set-associative instruction and data caches
 - 1.5-cycle cache array access, 3-cycle load-to-use latency
 - Pseudo-LRU (PLRU) replacement algorithm
 - Copy-back data cache that can function as a write-through cache on a page-by-page basis
 - Supports all Book E memory coherency modes
 - Supports EIS-defined cache-locking instructions, listed in Table 1-4.
- Dual-issue superscalar control
 - Two-instructions-per-clock peak issue rate
 - Precise exception handling
- Decode unit
 - 12-entry instruction queue (IQ)
 - Full hardware detection of interlocks
 - Decodes as many as two instructions per cycle

- Decode serialization control
- Register dependency resolution and renaming
- Branch prediction unit (BPU)
 - Dynamic branch prediction using a 512-entry, four-way set-associative branch target buffer (BTB), supported by the e500 BTB instructions listed in Table 1-1.
 - Branch prediction is handled in the fetch stages.
- Completion unit
 - As many as 14 instructions allowed in 14-entry completion queue (CQ)
 - In-order retirement of as many as two instructions per cycle
 - Completion and refetch serialization control
 - Synchronization for all instruction flow changes—interrupts, mispredicted branches, context-synchronizing instructions
- Issue queues
 - Two-entry branch instruction issue queue (BIQ)
 - Four-entry general instruction issue queue (BIQ)
- Branch unit—The branch unit (BU) is an execution unit and is distinct from the BPU. It executes (resolves) all branch and CR logical instructions.
- Two simple units (SU1 and SU2)
 - Single-cycle add and subtract
 - Single-cycle shift and rotate
 - Single-cycle logical operations
 - Support for 64-bit signal processing engine APU instructions in SU1
- Multiple-cycle unit (MU). The MU is shown in Figure 1-2. The MU has the following features:

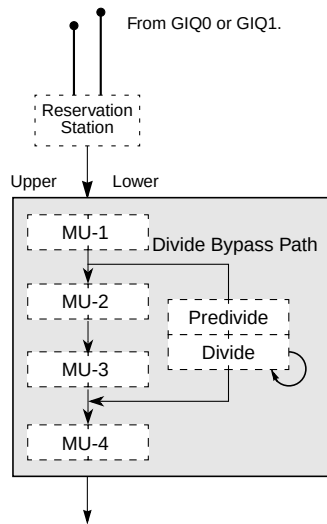


Figure 1-2. Four-Stage MU Pipeline, Showing the Divide Bypass

- Four-cycle latency for integer and SPE multiplication (including SPE vector and scalar, integer, fractional, and floating-point multiply instructions).
- Variable-latency divide: 4, 11, 19, and 35 cycles for all Book E and SPE integer divide instructions. If **rA** or **rB** is zero, a floating-point divide takes 4 cycles. All other cases take 29 cycles. Note that although most divide instructions take more than 4 cycles to execute, the MU allows subsequent multiply instructions to execute through all four MU stages in parallel with the divide.
- Four-cycle floating-point multiply
- Four-cycle floating-point add and subtract
- Load/store unit (LSU) is shown in Figure 1-3. The LSU has the following features:
 - 3-cycle load latency
 - Fully pipelined
 - Four-entry load miss queue allows up to four load misses before stalling
 - Load hits can continue to be serviced when the load miss queue is full
 - Seven-entry L1 store queue allows full pipelining of stores
 - The three-entry data line fill buffer is used for loads and cacheable stores. Stores are allocated here so loads can access data from the store immediately.
 - There are three entries in the data write buffer: one dedicated for snoop pushes, one dedicated for cast outs, and one that can be used for either.

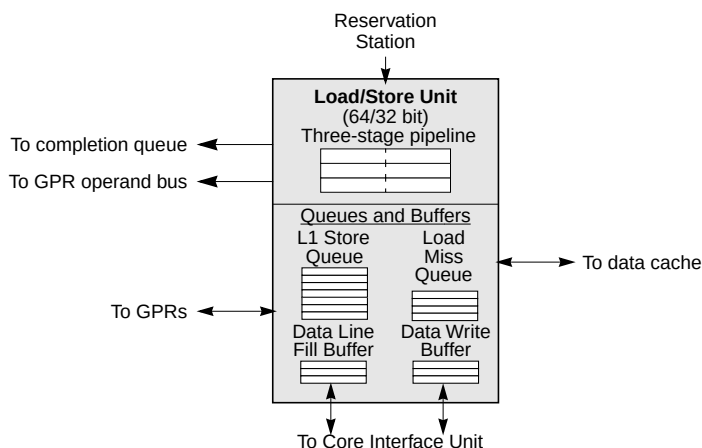


Figure 1-3. Three Stage Load/Store Unit

- Cache coherency
 - Supports both three-state MEI cache coherency and four-state cache coherency: modified-exclusive, exclusive, shared, and invalid (MESI)
 - Bus support for hardware-enforced coherency (bus snooping)
- Core complex bus (CCB)
 - High-speed, on-chip local bus with data tagging
 - 32-bit address bus
 - Address protocol with address pipelining and retry/copyback derived from bus used by previous generations of PowerPC processors (referred to as the 60x bus)
 - Two general-purpose read data, one write data bus
 - 128-bit data plus parity/tags (each data bus)
 - Support for out-of-order reads, in-order writes
 - Little to no data bus arbitration logic required for native systems
 - Easily adaptable to 60x-like environments
 - Support for one-level pipelining of addresses with address-retry responses
- Extended exception handling
 - Supports Book E interrupt model
 - Less than 10-cycle interrupt latency
 - Interrupt vector prefix register (IVPR)
 - Vector offset registers (IVORs) 0–15 as defined in Book E, plus e500-defined IVORs 32–35
 - Exception syndrome register (ESR)

Features

- Book E-defined preempting critical interrupt, including critical interrupt status registers (CSRR0 and CSRR1) and an **rfci** instruction
- e500-specific interrupts not defined in Book E architecture
 - SPE APU unavailable exception
 - Vector floating-point data exception
 - Vector floating-point round exception
 - Performance monitor
- Memory management unit (MMU)
 - Data L1 MMU
 - 4-entry, fully-associative TLB array for variable-sized pages
 - 64-entry, 4-way set-associative TLB for 4-Kbyte pages
 - Instruction L1 MMU
 - 4-entry, fully-associative TLB array for variable-sized pages
 - 64-entry, 4-way set-associative TLB for 4-Kbyte pages
 - Unified L2 MMU
 - 16-entry, fully-associative TLB array for variable-sized pages
 - 256-entry, 2-way set-associative TLB for 4-Kbyte pages
 - Software reload for TLBs
 - Virtual memory support for as much as 4 Gbytes (2^{32}) of virtual memory
 - Real memory support for as much as 4 Gbytes (2^{32}) of physical memory
 - Support for big-endian and true little-endian memory on a per-page basis
- Power management
 - Low power design
 - 1.5-volt design
 - Internal clock multipliers ranging from 1x to 8x from bus clock
 - Power-saving modes: standby and shutdown
 - Dynamic power management of execution units, caches, and MMUs
- Testability
 - LSSD scan design
 - JTAG interface
 - ESP support
 - ABIST for arrays
 - LBIST
 - Nexus debug support

- Reliability and serviceability
 - Internal code parity
 - Parity checking on e500 local bus

1.3 Instruction Set

The e500 implements the following instructions:

- The Book E instruction set for 32-bit implementations. This is composed primarily of the user-level instructions defined by the PowerPC user instruction set architecture (UISA). The e500 does not include the Book E floating-point instructions or the Load String Word Indexed instruction (**lswx**).
- The e500 supports the following implementation-specific instructions:
 - Instruction select APU. This APU consists of the Instruction Select instruction, **isel**, which incorporates an if...then else statement that selects between two source registers by comparison to a CR bit. This instruction eliminates conditional branches, decreases band latency, and reduces the code footprint.
 - Branch target buffer (BTB) locking APU instructions. The core complex provides a 512-entry BTB for efficient processing of branch instructions. The BTB is a branch target address cache (BTAC), organized as 128 rows with four-way set associativity, that holds the address and target instruction of the 512 most-recently taken branches. Table 1-1 lists the BTB instructions.

Table 1-1. BTB Locking APU Instructions

Name	Mnemonic	Syntax
Branch Buffer Load Entry and Lock Set	bblels	—
Branch Buffer Entry Lock Reset	bbelr	—

- SPE APU vector instructions. New vector instructions are defined that view the 64-bit GPRs as being composed of a vector of two 32-bit elements (some of the instructions also read or write 16-bit elements). These new instructions can also perform scalar operations by ignoring the results of the upper 32-bit half of the register file. Some scalar instructions are defined for DSP that produce a 64-bit scalar result. These instructions include vector floating-point instructions that operate on two 32-bit single-precision operands that reside in the upper and lower halves of the GPRs. For complete lists of SPE APU vector instructions, see Section 10.3.1.2, “SPE APU Instruction Model.”
- The SPE APU supports a subset of single-precision scalar floating-point instructions that use the lower 32-bits of the GPRs for single-precision floating-point calculations. This subset is referred to as the SPFP FPU. These instructions are listed in Table 1-2.

Table 1-2. SPE APU Scalar Floating-Point Instructions

Instruction	Mnemonic	Syntax
Floating-Point Absolute Value	efsabs	rD,rA
Floating-Point Add	efsadd	rD,rA,rB
Convert Floating-Point from Signed Fraction	efscfsf	rD,rB
Convert Floating-Point from Signed Integer	efscfsi	rD,rB
Convert Floating-Point from Unsigned Fraction	efscfuf	rD,rB
Convert Floating-Point from Unsigned Integer	efscfui	rD,rB
Floating-Point Compare Equal	efscmpeq	crfD,rA,rB
Floating-Point Compare Greater Than	efscmpgt	crfD,rA,rB
Floating-Point Compare Less Than	efscmplt	crfD,rA,rB
Convert Floating-Point to Signed Fraction	efscfsf	rD,rB
Convert Floating-Point to Signed Integer	efscfsi	rD,rB
Convert Floating-Point to Signed Integer with Round toward Zero	efscfsiz	rD,rB
Convert Floating-Point to Unsigned Fraction	efscfuf	rD,rB
Convert Floating-Point to Unsigned Integer	efscfui	rD,rB
Convert Floating-Point to Unsigned Integer with Round toward Zero	efscfui	rD,rB
Floating-Point Divide	efsddiv	rD,rA,rB
Floating-Point Multiply	efsmul	rD,rA,rB
Floating-Point Negative Absolute Value	efsnabs	rD,rA
Floating-Point Negate	efsneg	rD,rA
Floating-Point Subtract	efssub	rD,rA,rB
Floating-Point Test Equal	efststeq	crfD,rA,rB
Floating-Point Test Greater Than	efststgt	crfD,rA,rB
Floating-Point Test Less Than	efststlt	crfD,rA,rB

- Performance monitor APU—Table 1-3 lists performance monitor APU instructions.

Table 1-3. Performance Monitor APU Instructions

Name	Mnemonic	Syntax
Move from Performance Monitor Register	mfpmr	rD,PMRN
Move to Performance Monitor Register	mtpmr	PMRN,rS

- Cache block lock and unlock APU. The cache block lock and unlock APU consists of the instructions described in Table 1-4.

Table 1-4. Cache Block Lock and Unlock APU Instructions

Name	Mnemonic	Syntax
Data Cache Block Lock Clear	dcblc	CT, rA, rB
Data Cache Block Touch and Lock Set	dcbtls	CT, rA, rB
Data Cache Block Touch for Store and Lock Set	dcbtstls	CT, rA, rB
Instruction Cache Block Lock Clear	icblc	CT, rA, rB
Instruction Cache Block Touch and Lock Set	icbtls	CT, rA, rB

1.4 Instruction Flow

The e500 core is a pipelined, superscalar processor with parallel execution units that allow instructions to execute out of order but record their results in order. Pipelining breaks instruction processing into discrete stages, so multiple instructions in an instruction sequence can occupy the successive stages: as an instruction completes one stage, it passes to the next, leaving the previous stage available to a subsequent instruction. So, even though it may take multiple cycles for one instruction to pass through all of the pipeline stages, once a pipeline is full, instructions complete with a throughput that is far shorter than the latency.

A superscalar processor is one that issues multiple independent instructions into separate execution units, allowing parallel execution. The e500 core has five execution units, one each for branch (BU), load/store (LSU), and multiple-cycle operations (MU), and two for simple arithmetic (SU1, and SU2). The MU and SU1 arithmetic execution units also execute 64-bit SPE vector instruction, using both the lower and upper halves of the 64-bit GPRs.

The parallel execution units allow multiple instructions to execute in parallel and out of order. For example, a low-latency addition instruction that is issued to an SU after an integer divide is issued to the MU should finish executing before the higher latency divide instruction. The add instruction can make its results available to a subsequent instruction, but it cannot update the architected GPR specified as its target operand ahead of the multiple-cycle divide instruction.

1.4.1 Branch Detection and Prediction

To improve branch performance, the e500 provides an implementation-specific dynamic branch prediction using a branch target buffer (BTB) to resolve branch instructions and improve the accuracy of branch predictions. Each of the 512 entries in the four-way set associative address cache of branch target addresses includes a 2-bit branch history table, whose value is incremented or decremented depending on whether the branch was taken. These bits can take four values indicating strongly taken, weakly taken, weakly not taken, and strongly not taken.

The BTB is used not only to predict branches, but to detect branches during the fetch stage, offering an efficient way to access instruction streams for branches predicted as taken, to accurately predict those branches, to speculatively execute speculative instructions, and to quickly recover and revise incorrect predictions.

In the e500, all branch instructions are assigned positions in the completion queue at dispatch and speculative instructions in branch target streams are allowed to execute and proceed through the completion queue, although they can complete only after the branch prediction is resolved as correct and after the branch instruction itself completes.

If a branch resolves as correct, instructions in the target stream are marked nonspeculative and are allowed to complete. If the branch history bits in the BTB had indicated weakly taken or weakly not taken, the prediction is upgraded to strongly taken or strongly not taken.

If a branch resolves as incorrect, instructions in the target stream are flushed from the execution pipeline, the branch history bits are update in the BTB entry, and nonspeculative fetching begins from the correct path.

1.4.2 e500 Execution Pipeline

The seven stages of the e500 execution pipeline—fetch, fetch2/predecode, decode/dispatch, issue, execute, complete, and write back—are highlighted in grey in Figure 1-4.

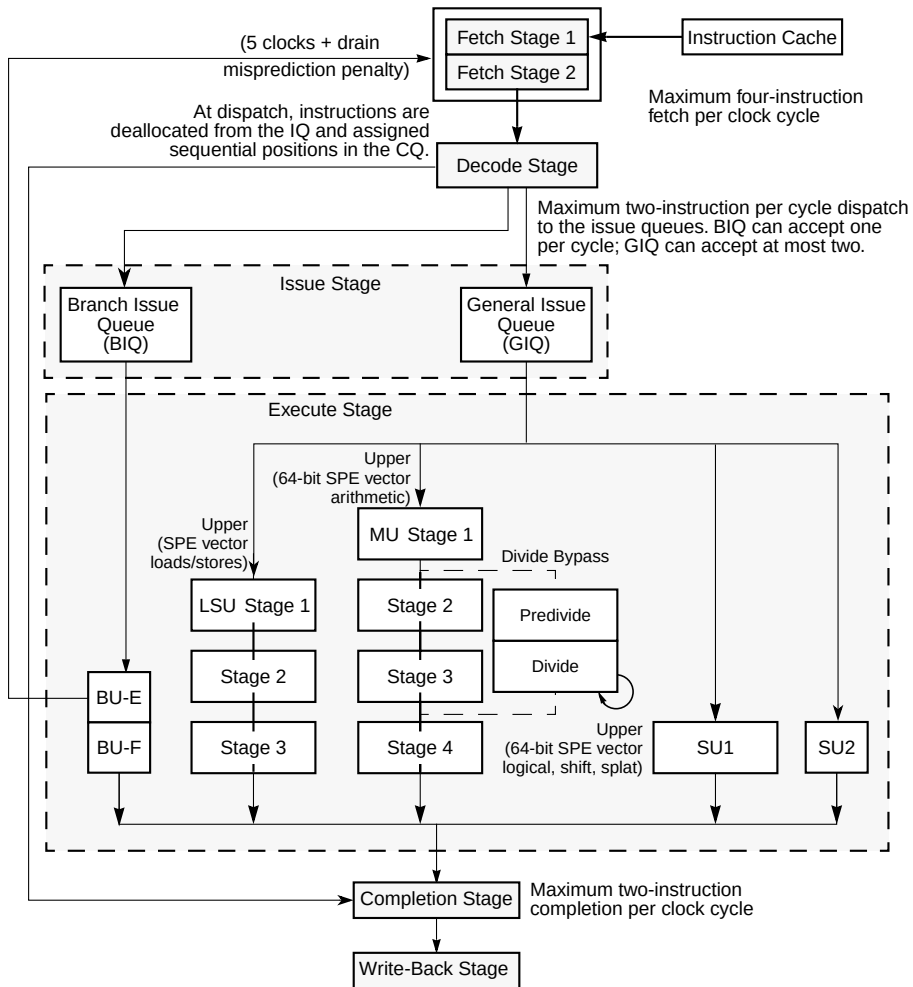


Figure 1-4. Instruction Pipeline Flow

The common pipeline stages are as follows:

- **Instruction fetch**—Includes the clock cycles necessary to request an instruction and the time the memory system takes to respond to the request. Instructions retrieved are latched into the instruction queue (IQ) for subsequent consideration by the dispatcher.

Instruction fetch timing depends on many variables, such as whether an instruction is in the on-chip instruction cache or an L2 cache (if implemented). Those factors increase when it is necessary to fetch instructions from system memory and include

the processor-to-bus clock ratio, the amount of bus traffic, and whether any cache coherency operations are required.

Because there are so many variables, unless otherwise specified, the instruction timing examples in this chapter assume optimal performance and show the portion of the fetch stage in which the instruction is in the instruction queue. The fetch1 and fetch2 stages are primarily involved in retrieving instructions.

- The decode/dispatch stage fully decodes each instruction; most instructions are dispatched to the issue queues (**isync**, **rfl**, **sc**, **nops**, and some other instructions do not go to issue queues).
- The two issue queues, BIQ and GIQ, can accept as many as one and two instructions, respectively, in a cycle. Instruction dispatch requires the following:
 - Instructions dispatch only from the two lowest IQ entries—IQ0 and IQ1.
 - A total of two instructions can be dispatched to the issue queues per clock cycle.
 - Space must be available in the CQ for an instruction to decode and dispatch (this includes instructions that are assigned a space in the CQ but not in an issue queue).

Dispatch is treated as an event at the end of the decode stage. The issue stage reads source operands from rename registers and register files and determines when instructions are latched into the execution unit reservation stations. Note that the e500 has 14 rename registers, one for each completion queue entry, so instructions cannot stall because of a shortage of rename registers.

The general behavior of the two issue queues is described as follows:

- The GIQ accepts as many as two instructions from the dispatch unit per cycle. SU1, SU2, MU, and all LSU instructions (including SPE APU loads and stores) are dispatched to the GIQ, shown in Figure 1-5.

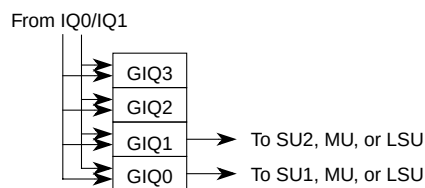


Figure 1-5. GPR Issue Queue (GIQ)

Instructions can be issued out-of-order from the bottom two GIQ entries (GIQ1–GIQ0). GIQ0 can issue to SU1, MU, and LSU. GIQ1 can issue to SU2, MU, and LSU.

Note that SU2 executes a subset of the instructions that can be executed in SU1. The ability to identify and dispatch instructions to SU2 increases the availability of SU1 to execute more computational-intensive instructions.

An instruction in GIQ1 destined for SU2 or the LSU need not wait for an MU instruction in GIQ0 that is stalled behind a long-latency divide.

- The execute stage accepts instructions from its issue queue when the appropriate reservation stations are not busy. In this stage, the operands assigned to the execution stage from the issue stage are latched.

The execution unit executes the instruction (perhaps over multiple cycles), writes results on its result bus, and notifies the CQ when the instruction finishes. The execution unit reports any exceptions to the completion stage. Instruction-generated exceptions are not taken until the excepting instruction is next to retire.

Most integer instructions have a 1-cycle latency, so results of these instructions are available 1 clock cycle after an instruction enters the execution unit. The MU and LSU are pipelined, as shown in Figure 1-4.

- The complete and write-back stages maintain the correct architectural machine state and commit results to the architecture-defined registers in the proper order. If completion logic detects an instruction containing an exception status or a mispredicted branch, all following instructions are cancelled, their execution results in rename registers are discarded, and the correct instruction stream is fetched.

The complete stage ends when the instruction is retired. Two instructions can be retired per clock cycle. If no dependencies exist, as many as two instructions are retired in program order. Section 4.7.4, “Completion Unit Resource Requirements” describes completion dependencies.

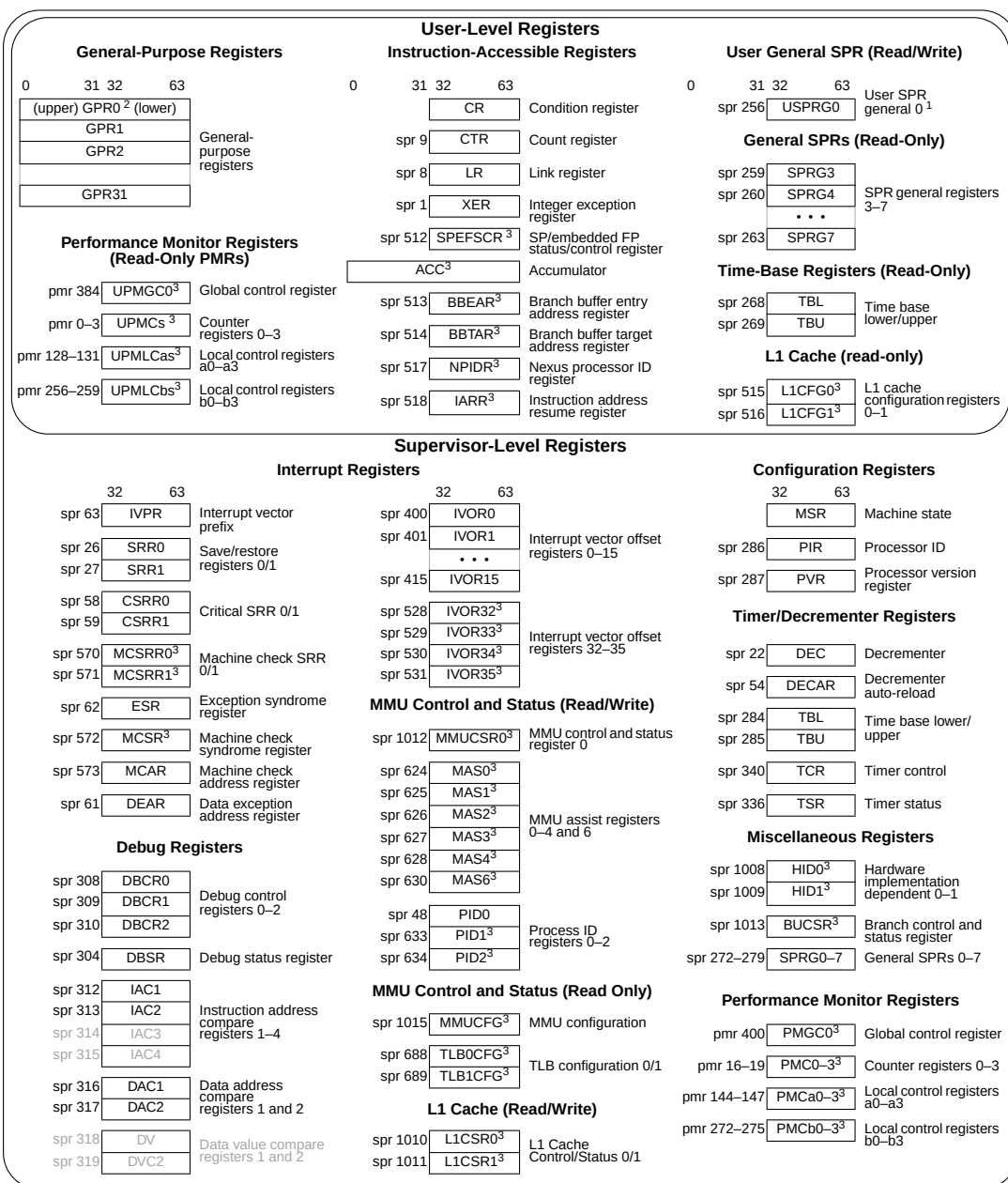
The write-back stage occurs in the clock cycle after the instruction is retired.

The e500 core also provides new instructions that perform single-instruction, multiple-data (SIMD) operations. These signal processing instructions consist of parallel operations on both the upper and lower 32 bits of two 64-bit GPR values and produce two 32-bit results written to a 64-bit GPR.

As shown in Figure 1-4, the LSU, MU, and SU1 replicate logic to support the 64-bit vector operations. Although a vector instruction generates separate, discrete results in the upper and lower halves of the target GPR, latency and throughput for vector instructions are the same as those for their scalar equivalents.

1.5 Programming Model

The following section describes the e500 core registers defined in Book E, the Motorola Book E implementation standards (EIS), and registers that are specific to the e500. Table 1-6 shows the e500 register set.



¹ USPRG0 is a separate physical register from SPRG0.

² The 64-bit GPR registers are accessed by the SPE as separate 32-bit registers by SPE instructions. Only SPE vector instructions can access the upper word.

³ These registers are e500-specific and are not part of the Book E architecture.

Figure 1-6. e500 Core Programming Model

1.6 On-Chip Cache Implementation

The core complex contains separate 32-Kbyte, eight-way set-associative, level 1 (L1) instruction and data caches to give rapid access to instructions and data.

Data caches support three-state MEI and four-state MESI memory coherency protocols with 3-bit status and 1-bit coherency valid fields. The core complex broadcasts all cache management functions based on the setting of the address broadcast enable bit, HID1[ABE], allowing management of other caches in the system.

The caches implement a pseudo-least-recently-used (PLRU) replacement algorithm.

Parity generation and checking may be enabled for both caches, and each cache can be independently invalidated through L1CSR1 and L1CSR0. Additionally, instructions are provided to perform cache locking and unlocking on both data and instruction caches on a cache-block granularity. These are listed in Section 1.9.3, “Cache Control Instructions.”

1.7 Interrupts and Exception Handling

The e500 core supports an extended exception handling model, with nested interrupt capability and extensive interrupt vector programmability. The following sections define the exception model, including an overview of exception handling as implemented on the e500 core, a brief description of the exception classes, and an overview of the registers involved in the processes.

1.7.1 Exception Handling

In general, interrupt processing begins with an exception that occurs due to external conditions, errors, or program execution problems. When the exception occurs, the processor checks to verify interrupt processing is enabled for that particular exception. If enabled, the interrupt causes the state of the processor to be saved in the appropriate registers, and prepares to begin execution of the handler located at the associated vector address for that particular exception.

Once the handler is executing, the implementation may need to check one or more bits in the exception syndrome register (ESR) or the SPEFSCR, depending on the exception type, to verify the specific cause of the exception and take appropriate action.

The core complex provides the interrupts described in Section 1.7.5, “Interrupt Registers,” and Table 1-7.

The e500 core processes all interrupts as either machine check, critical, or non-critical types. Separate control and status register sets are provided for each type of interrupt. The core handles interrupts from these three categories in the following priority order:

1. Machine check—highest priority
2. Critical interrupt—medium priority
3. Noncritical interrupt—lowest priority

1.7.2 Interrupt Classes

All interrupts may be categorized as asynchronous/synchronous and critical/noncritical.

- Asynchronous interrupts are caused by events that are independent of instruction execution. For asynchronous interrupts, the address reported in the save/restore register is the address of the instruction that would have executed next had the asynchronous interrupt not occurred.
- Synchronous interrupts are those that are caused directly by the execution or attempted execution of instructions. Synchronous inputs are further divided into precise and imprecise types.
 - Synchronous precise interrupts are those that precisely indicate the address of the instruction causing the exception that generated the interrupt or, in some cases, the address of the immediately following instruction. The interrupt type and status bits allow determination of which of the two instructions has been addressed in the appropriate save/restore register.
 - Synchronous imprecise interrupts are those that may indicate the address of the exception causing the exception that generated the interrupt, or some instruction after the instruction causing the interrupt. If the interrupt was caused by either the context synchronizing mechanism, or the execution synchronizing mechanism, the address in the appropriate save/restore register is the address of the interrupt forcing instruction. If the interrupt was not caused by either of those mechanisms, the address in the save/restore register is the last instruction to start execution, and may not have completed. No instruction following the instruction in the save/restore register has executed.

1.7.3 Interrupt Types

There are three types of interrupt, described as follows:

- Noncritical interrupts—First-level interrupts that allow the processor to change program flow to handle conditions generated by external signals, errors or unusual

conditions arising from program execution, or from programmable timer-related events.

These interrupts are largely identical to those previously defined by OEA portion the Power PC architecture. They use save and restore registers (SRR0/SRR1) to save state when they are taken and they use the **rfi** instruction to restore state.

Asynchronous non-critical interrupts can be masked by the external interrupt enable bit, MSR[EE].

- **Critical interrupts**—Critical interrupts can be taken during a non-critical interrupt or during regular program flow. They use the critical save and restore registers (CSRR0/CSRR1) to save state when they are taken and they use the **rfci** instruction to restore state. These interrupts can be masked by the critical enable bit, MSR[CE]. Book E defines the critical input, watchdog timer, and machine check interrupts as critical interrupts, but the e500 defines a third set of resources for the machine check interrupt, as described below.
- **Machine check interrupt**—The e500 defines a separate set of resources for the machine check interrupt. They use the machine check save and restore registers (MCSRR0/MCSRR1) to save state when they are taken and they use the **rfmci** instruction to restore state. These interrupts can be masked by the machine check enable bit, MSR[ME].

All interrupts except machine check are ordered within the two categories of noncritical and critical, such that only one interrupt of each category is reported, and when it is processed (taken), no program state is lost. Because save/restore register pairs are serially reusable, program state may be lost when an unordered interrupt is taken (see Section 5.10, “Interrupt Ordering and Masking”).

1.7.4 Upper Bound on Interrupt Latencies

The core complex interrupt latency is defined here as the number of core clocks between the sampling of the interrupt signal as asserted and the initiation of the IVOR fetch (that is, the fetch of the first instruction in the handler). The interrupt latency of the core complex is determinate unless a guarded load or a cache-inhibited **stwcx.** instruction is being executed. As a guarded load or a cache-inhibited **stwcx.** instruction is being executed, the interrupt latency of the core complex is indeterminate. The minimum interrupt latency for the core complex is 3 core clocks and the maximum is 8. These clocks are described in Table 1-5.

Table 1-5. Interrupt Latency

Action	Guarded Window	Reaching a Recoverable State.	Interrupt Processing	Redirect
Clocks	4/0	2/1	1	1

The interrupt latency defined in Table 1-5 does not include the 2 bus clock cycles required to synchronize the interrupt signal from the pad.

When ever an interrupt is taken, all the instructions in the instruction queue are thrown away except when the oldest instruction is a load/store instruction. That is, if an asynchronous interrupt is being serviced and the oldest instruction is not a load/store instruction, the core complex goes straight from sampling the interrupt to ensuring a recoverable state and issuing an exception. If a load/store instruction is oldest, the core complex waits 4 clocks before ensuring a recoverable state. During this time, any instruction finished by the load/store unit is deallocated.

1.7.5 Interrupt Registers

The registers associated with interrupt and exception handling are described in Table 1-6.

Table 1-6. Interrupt Registers

Register	Description
Non-Critical Interrupt Registers	
SRR0	Save/restore registers—Stores the address of the instruction causing the exception or the address of the instruction that will execute after the rfi instruction.
SRR1	Save/restore register 1—SRR1 saves machine state on non-critical interrupts and restores machine state after an rfi instruction is executed.
Critical Interrupt Registers	
CSRR0	Critical save/restore register 0—on critical interrupts, CSRR0 stores either the address of the instruction causing the exception or the address of the instruction that will execute after the rfci instruction.
CSRR1	Critical save/restore register 1—CSRR1 saves machine state on critical interrupts and restores machine state after an rfci instruction is executed.
Machine Check Interrupt Registers	
MCSRR0	Machine check save/restore register 0—MCSRR0 is used to store the address of the instruction that will execute after an rfmci instruction is executed.
MCSRR1	Machine check save/restore register 1—MCSRR1 stores machine state on machine check interrupts and restores machine state (if recoverable) after an rfmci instruction is executed.
MCAR	Machine check address register —MCAR holds the address of the data or instruction that caused the machine check interrupt. MCAR contents are not meaningful if a signal triggered the machine check interrupt.
Syndrome Registers	
MCSR	Machine check syndrome register—MCSR saves machine state information on machine check interrupts and restores machine state after an rfmci instruction is executed.
ESR	Exception syndrome register—ESR provides a syndrome to differentiate between the different kinds of exceptions that generate the same interrupt type. Upon generation of a specific exception type, the associated bit is set and all other bits are cleared.
SPE APU Interrupt Registers	
SPEFSCR	Signal processing and embedded floating-point status and control register—provides interrupt control and status, as well as various condition bits associated with the operations performed by the SPE APU.

Table 1-6. Interrupt Registers (continued)

Register	Description
Other Interrupt Registers	
DEAR	Data exception address register—DEAR contains the address that was referenced by a load, store, or cache management instruction that caused an alignment, data TLB miss, or data storage interrupt.
IVPR	Interrupt vector prefix register—IVPR[32–47] contains the high-order 16 bits of the address of the exception processing routines defined in the IVOR registers.
IVORs	Interrupt vector offset registers—The IVORs contain the low-order offset of the address of the exception processing routines defined in the IVOR registers. See Table 1-7.

Each interrupt has an associated interrupt vector address, obtained by concatenating the IVPR value with the address index in the associated IVOR (that is, IVPR[32–47]||IVOR n [48–59]||0b0000). The resulting address is that of the instruction to be executed when that interrupt occurs. IVPR and IVOR values are indeterminate on reset, and must be initialized by the system software using **mtspr**. Table 1-7 lists IVOR registers implemented on the e500 and the associated interrupts. For more information, see Chapter 5, “Interrupts and Exceptions.”

Table 1-7. Interrupt Vector Registers and Exception Conditions

Register	Interrupt
Book E–Defined IVORs	
IVOR0	Critical input
IVOR1	Machine check interrupt offset
IVOR2	Data storage interrupt offset
IVOR3	Instruction storage interrupt offset
IVOR4	External input interrupt offset
IVOR5	Alignment interrupt offset
IVOR6	Program interrupt offset
IVOR7	Floating-point unavailable interrupt offset (not supported on the e500)
IVOR8	System call interrupt offset
IVOR9	Auxiliary processor unavailable interrupt offset (not supported on the e500)
IVOR10	Decrementer interrupt offset
IVOR11	Fixed-interval timer interrupt offset
IVOR12	Watchdog timer interrupt offset
IVOR13	Data TLB error interrupt offset
IVOR14	Instruction TLB error interrupt offset
IVOR15	Debug interrupt offset
e500-Specific IVORs	

Table 1-7. Interrupt Vector Registers and Exception Conditions (continued)

Register	Interrupt
IVOR32	SPE APU unavailable interrupt offset
IVOR33	SPE floating-point data exception interrupt offset
IVOR34	SPE floating-point round exception interrupt offset
IVOR35	Performance monitor

1.8 Memory Management

The e500 core complex supports demand-paged virtual memory as well other memory management schemes that depend on precise control of effective-to-physical address translation and flexible memory protection as defined by Book E. The mapping mechanism consists of software-managed TLBs that support variable-sized pages with per-page properties and permissions. The following properties can be configured for each TLB:

- User mode page execute access
- User mode page read access
- User mode page write access
- Supervisor mode page execute access
- Supervisor mode page read access
- Supervisor mode page write access
- Write-through required (W)
- Caching inhibited (I)
- Memory coherence required (M)
- Guarded (G)
- Endianess (E)
- User-definable (U0–U3), a 4-bit implementation-specific field

The core complex employs a two-level memory management unit (MMU) architecture. There are separate instruction and data level 1 (L1) MMUs backed up by a unified level-2 (L2) MMU, as shown in Figure 1-7.

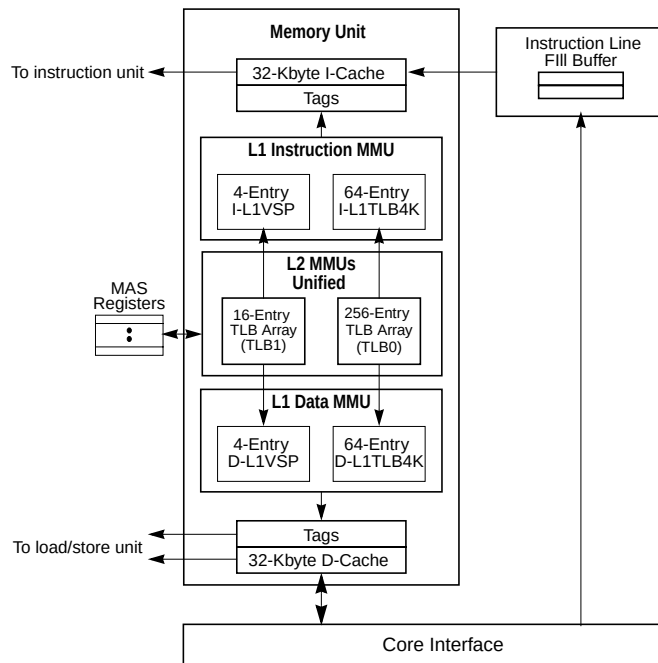


Figure 1-7. MMU Structure

The level-1 MMUs have the following features:

- Four-entry, fully associative TLB array that supports all nine page sizes
- 64-entry, 4-way set-associative TLB 4-Kbyte array that supports 4-Kbyte pages only
- Hardware partially managed by L2 MMU
- Supports snooping of TLBs by both internal and external **tlbivax** instructions

The level-2 MMU has the following features:

- A 16-entry, fully associative L2 TLB array (TLB1) that supports all nine page sizes
- 256-entry, 2-way set-associative TLB array (TLB0) that supports only 4-Kbyte pages
- Hardware assist for TLB miss exceptions
- Software managed by **tlbre**, **tlbwe**, **tlbsx**, **tlbsync**, **tlbivax**, and **mtspr** instructions
- Supports snooping of TLB by both internal and external **tlbivax** instructions

1.8.1 Address Translation

The core complex fetch and load/store units generate 32-bit effective addresses. The MMU translates these addresses to real 32-bit addresses (which are used for memory bus accesses) using an interim 41-bit virtual address. The translation flow is shown in Figure 1-8.

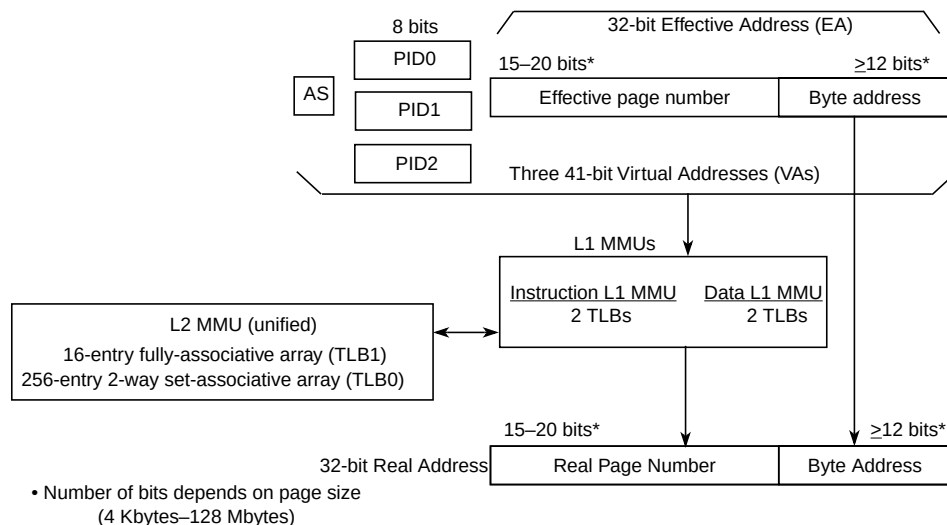


Figure 1-8. Effective-to-Real Address Translation Flow

The appropriate L1 MMU (instruction or data) is checked for a matching address translation. The instruction L1 MMU and data L1 MMU operate independently and can be accessed in parallel, so that hits for instruction accesses and data accesses can occur in the same clock. If an L1 MMU misses, the request for translation is forwarded to the unified (instruction and data) L2 MMU. If found, the contents of the TLB entry are concatenated with the byte address to obtain the physical address of the requested access. On misses, the L1 TLB entries are replaced from their L2 TLB counterparts using a true LRU algorithm.

1.8.2 MMU Assist Registers (MAS1–MAS4 and MAS6)

Book E defines SPR numbers for the MMU assist registers, which are used to hold values either read from or to be written to the TLBs and information required to identify the TLB to be accessed. Book E leaves MAS register bit definitions up to the implementations. To ensure consistency among Motorola Book E processors, certain aspects of the implementation are defined by the Motorola Book E standard, whereas more specific details are left to individual implementations. MAS3 implements the real page number (RPN), the user attribute bits, U0–U3, and permission bits (UX, SX, UW, SW, UR, SR) that specify user and supervisor read, write, and execute permissions.

The e500 does not implement MAS5.

MAS registers are affected by the following instructions:

- MAS registers are accessed with the **mtspr** and **mfspr** instructions.
- The TLB Read Entry instruction (**tlbre**) causes the contents of a single TLB entry from the L2 MMU to be placed in defined locations in MAS0–MAS3. The TLB entry to be extracted is determined by information written to MAS0 and MAS2 before the **tlbre** instruction is executed.
- The TLB Write Entry instruction (**tlbwe**) causes the information stored in certain locations of MAS0–MAS3 to be written to the TLB specified in MAS0.
- The TLB Search Indexed instruction (**tlbsx**) updates MAS registers conditionally, based on success or failure of a lookup in the L2 MMU. The lookup is specified by the instruction encoding and specific search fields in MAS6. The values placed in the MAS registers may differ, depending on a successful or unsuccessful search.

For TLB miss and certain MMU-related DSI/ISI exceptions, MAS4 provides default values for updating MAS0–MAS2.

1.8.3 Process ID Registers (PID0–PID2)

The e500 core complex also implements three process ID (PID) registers that hold the values used to construct the three virtual addresses for each access. These process IDs provide an extended page sharing capability. Which of these three virtual addresses is used is controlled by the TID field of a matching TLB entry, and when TID = 0x00 (identifying a page as globally shared), the PID values are ignored.

A hit to multiple TLB entries in the L1 MMU (even if they are in separate arrays) or a hit to multiple entries in the L2 MMU is considered to be a programming error.

1.8.4 TLB Coherency

The core complex provides the ability to invalidate a TLB entry as defined in the Book E architecture. The **tlbivax** instruction invalidates a matching local TLB entry. Execution of this instruction is also broadcast on the core complex bus (CCB) if HID1[ABE] = 1. The core complex also snoops TLB invalidate transactions on the CCB from other bus masters.

1.9 Memory Coherency

The core complex supports both three- and four-state memory coherency. Memory coherency is hardware-supported on the system bus through bus snooping and the retry/copyback bus protocol, and through broadcasting of cache management instructions. Translation coherency is also hardware-supported through broadcasting and bus snooping of TLB invalidate transactions. The three-state MEI coherency protocol is suited to

single-processor systems that include DMA and is also applicable to multiple-processor systems with low to moderate real-time data sharing. The four-state MESI protocol supports efficient large-scale real-time data sharing between multiple caching bus masters. The core complex can be configured to default to three- or four-state operation. Additionally, the three- or four-state operation may be selected on a page-by-page basis for system optimization.

1.9.1 Atomic Update Memory References

The e500 core supports atomic update memory references for both aligned word forms of data using the load and reserve and store conditional instruction pair, **lwarx** and **stwcx..** Typically, a load and reserve instruction establishes a reservation and is paired with a store conditional instruction to achieve the atomic operation. However, there are restrictions and requirements for this functionality. It is the programmer's responsibility to preserve reservations across context switches and to protect reservations in multiprocessor implementations.

1.9.2 Memory Access Ordering

To optimize performance the core complex supports weakly ordered references to memory. Thus the e500 manages the order and synchronization of instructions to insure proper execution when memory is shared between multiple processes or programs. The cache and data memory control attributes along with **msync** and **mbar** provide the required access control; **msync** and **mbar** are also broadcast on the CCB to provide the appropriate control in the case of multiprocessor or shared memory systems.

1.9.3 Cache Control Instructions

The core complex supports Book E instructions for performing a full range of cache control functions, including cache locking by line. The core complex supports broadcasting and snooping of these cache control instructions on the CCB. The e500 core also supports the following e500-specific cache locking instructions:

- Data Cache Block Lock Clear (**dcblc**)
- Data Cache Block Touch-and-Lock Set (**dcbtls**)
- Data Cache Block Touch for Store and Lock Set (**dcbtstls**)
- Instruction Cache Block Lock Clear (**icblc**)
- Instruction Cache Block Touch-and-Lock Set (**icbtls**)

1.9.4 Programmable Page Characteristics

Cache and memory attributes are programmable on a per-page basis. In addition to the write-through, caching-inhibited, memory coherency enforce, and guarded characteristic

defined by the WIMG bits, Book E defines an endianness bit that allows selection of big- or little-endian byte ordering on a per-page basis.

1.10 Core Complex Bus (CCB)

The core complex defines a versatile local bus interface that allows a wide range of system performance and system-complexity trade-offs. The interface defines the following buses.

- An address-out bus for mastering bus transactions
- An address-in bus for snooping internal resources
- Three tagged data buses

Two of the data buses are general-purpose data-in buses for reads, and the third is a data-out bus for writes. The two data-in buses feature support for out-of-order read transactions from two different sources simultaneously, and all three data buses may be operated concurrently. The address-in bus supports snooping for external management of the L1 caches and TLBs by other bus masters. The core complex broadcasts and snoops the cache and TLB management instructions accordingly. It is envisioned that a wide range of system implementations can be constructed from the defined interface.

1.11 Performance Monitoring

The e500 core provides a performance monitoring capability that allows counting of events such as processor clocks, instruction cache misses, data cache misses, mispredicted branches, and others. The count of these events may be configured to trigger a performance monitor exception following the e500 interrupt model. This interrupt is assigned to vector offset register IVOR35.

The register set associated with the performance monitoring function consists of counter registers, a global control register, and local control registers. These registers are read/write from supervisor mode, and each register is reflected to a corresponding read-only register for user mode. Two instructions, **mtpmr** and **mfpmr**, are provided for moving data to and from these registers. An overview of the performance monitoring registers is provided in the following sections.

1.11.1 Global Control Register

The PMGC0 register provides global control of the performance monitoring facility from supervisor mode. From this register all counters may be frozen, unfrozen, or configured to freeze on an enabled condition or event. Additionally, the performance monitoring facility may be disabled or enabled from this register. The contents of PMGC0 are reflected to UPMGC0 which may be read from user mode using the **mfpmr** instruction.

1.11.2 Performance Monitor Counter Registers

There are four counter registers (PCM0–PCM3) provided in the performance monitoring facility. These 32-bit registers hold the current count for software selectable events, and can be programmed to generate an exception on overflow. These registers may be written or read from supervisor mode using the **mtpmr** and **mfpmr** instructions. The contents of these registers are reflected to UPCM0–UPCM3, which can be read from user mode with **mfpmr**.

The exception generated on overflow can be masked by clearing MSR[EE].

1.11.3 Local Control Registers

For each of the counter registers, there are two corresponding local control registers. These two registers specify which of the 128 available events is to be counted, what specific action is to be taken on overflow, and various options for freezing a counter value under given modes or conditions.

- PMLCa0–PMLCa3 provide fields that allow freezing of the corresponding counter in user mode, supervisor mode, or under software control. Additionally, the overflow condition may be enabled or disabled from this register. The contents of these registers are reflected to UPMCLa0–UPMLCa3, which can be read from user mode with **mfpmr**.
- PMLCb0–PMLCb3 provide count scaling for each counter register using configurable threshold and multiplier values. The threshold is a 6-bit value and the multiplier is a 3-bit encoded value, allowing 8 multiplier values in the range of 1 to 128. Any counter may be configured to increment only when an event occurs more than [threshold x multiplier] times. The contents of these registers are reflected to UPMCLb0–UPMLCb3, which can be read from user mode with **mfpmr**.

1.12 Legacy Support of PowerPC Architecture

This section provides an overview of the architectural differences and compatibilities of the e500 core compared with the AIM PowerPC architecture. The two levels of the e500 programming environment are as follows:

- User level—This defines the base user-level instruction set, user-level registers, data types, memory conventions, and the memory and programming models seen by application programmers.
- Supervisor level—This defines supervisor-level resources typically required by an operating system, the memory management model, supervisor level registers, and the exception model.

In general, the e500 core supports the user-level architecture from the existing AIM architecture. The following subsections are intended to highlight the main differences. For specific implementation details refer to the relevant chapter.

1.12.1 Instruction Set Compatibility

The following sections generally describe the user and supervisor instruction sets.

1.12.1.1 User Instruction Set

The e500 core executes legacy user-mode binaries and object files with the following exceptions.

- The e500 supports only single-precision floating-point operations, with different encoding than the AIM definition of the PowerPC architecture. Additionally, the e500 core uses GPRs for floating-point operations, rather than the FPRs defined by the UISA. Most porting of floating-point operations can be handled by recompiling; however, there are new instructions specific to the e500 SPE APU.
- String instructions are not implemented on the e500; therefore, trap emulation must be provided to ensure backward compatibility.

1.12.1.2 Supervisor Instruction Set

The supervisor mode instruction set defined by the AIM version of the PowerPC architecture is compatible with the e500 with the following exceptions:

- The MMU architecture is different, so some TLB manipulation instructions have different semantics.
- Instructions that support the BATs and segment registers are not implemented.

1.12.2 Memory Subsystem

Both Book E and the AIM version of the PowerPC architecture provide separate instruction and data memory resources. The e500 provides additional cache control features, including cache locking.

1.12.3 Exception Handling

Exception handling is generally the same as that defined in the AIM version of the PowerPC architecture for the e500, with the following differences:

- Book E defines a new critical interrupt, providing an extra level of interrupt nesting. The critical interrupt includes external critical and watchdog timer time-out inputs.
- The e500 implements the machine check exception differently from the Book E and from the AIM definition. It defines the Return from Machine Check Interrupt instruction, **rfmci**, and two machine check save/restore registers, MCSRR0 and MCSRR1.
- Book E processors can use IVPR and IVORs to set exception vectors individually, but they can be set to the address offsets defined in the OEA to provide compatibility.

- Unlike the AIM version of the PowerPC architecture, Book E does not define a reset vector; execution begins at a fixed virtual address, 0xFFFF_FFFC.
- Some Book E and e500-specific SPRs are different from those defined in the AIM version of the PowerPC architecture, particularly those related to the MMU functions. Much of this information has been moved to a new exception syndrome register (ESR).
- Timer services are generally compatible, although Book E defines a new decremter auto reload feature and the fixed-interval timer critical interrupt, which are implemented in the e500 core.

An overview of the interrupt and exception handling capabilities of the e500 core can be found in Section 1.7, “Interrupts and Exception Handling.”

1.12.4 Memory Management

The e500 core implements a straightforward virtual address space that complies with the Book E MMU definition, eliminating segment registers and block address translation resources. Book E defines resources for fixed 4-Kbyte pages and multiple, variable page sizes that can be configured in a single implementation. TLB management is provided with new instructions and SPRs.

1.12.5 Reset

Book E-compliant cores do not share a common reset vector with the AIM version of the PowerPC architecture. Due to the additional features and capabilities of the e500 core, the initialization process is different. Unlike the AIM version of the PowerPC core, once execution begins, the e500 core is in virtual mode with a hardware-initialized TLB entry.

1.12.6 Other Differences

Unlike the AIM version of the PowerPC architecture, where little-endian mode is controlled on a system basis, Book E allows control of endian mode on a memory page basis. Additionally, the e500 core effects true little-endian mode by byte swapping.

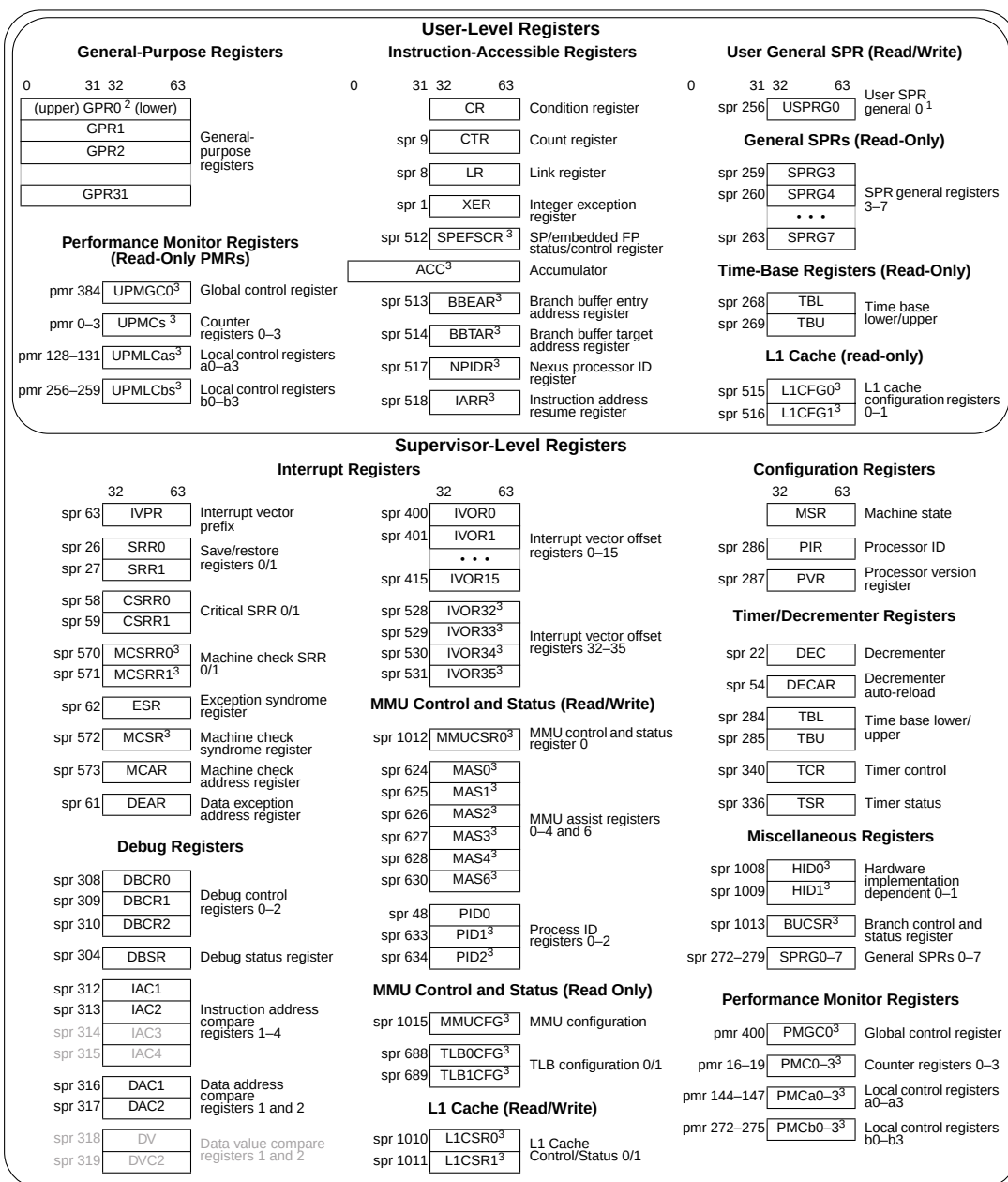
Chapter 2

Register Model

The e500 core complex is a 32-bit implementation of the Book E PowerPC architecture, which includes a recognition that different processor implementations may require extensions or deviations from the architectural descriptions. This chapter provides additional information about the Book E architecture as it relates specifically to the e500 core complex. Specific details on how registers are accessed are provided in Chapter 3, “Instruction Model.”

2.1 e500 Register Model

The following sections describe the e500 core register model as defined in Book E and the additional implementation-specific registers unique to the e500 core. Table 2-1 shows the e500 register set.



¹ USPRG0 is a separate physical register from SPRG0.

² The 64-bit GPR registers are accessed by the SPE as separate 32-bit registers by SPE instructions. Only SPE vector instructions can access the upper word.

³ These registers are e500-specific and are not part of the Book E architecture.

Figure 2-1. e500 Core Programming Model

2.2 Register Model for 32-Bit Book E Implementations

Book E processors implement the following three types of software-accessible registers:

- Book E-defined registers that are accessed as part of instruction execution. These include the following:
 - The following registers are used for integer operations:
 - General-purpose registers (GPRs)—Book E defines a set of 32 GPRs used to hold source and destination operands for load, store, arithmetic, and computational instructions, and to read and write to other registers.
 - Integer exception register (XER)—Bits in this register are set based on the operation of an instruction considered as a whole, not on intermediate results (for example, the Subtract from Carrying instruction (**subfc**), the result of which is specified as the sum of three values, sets bits in the XER based on the entire operation, not on an intermediate sum).

These registers are described in Section 2.3, “Registers for Integer Operations.”

- Condition register (CR)—Used to record conditions such as overflows and carries that occur as a result of executing arithmetic instructions (including those implemented by the SPE APU). The CR is described in Section 2.4, “Registers for Branch Operations.”
- Machine state register (MSR)—Used by the operating system to configure parameters such as user-/supervisor mode, address space, and enabling of asynchronous interrupts. This register is described in Section 2.5.1, “Machine State Register (MSR),” grouped with processor control SPRs.
- Book E-defined special-purpose registers (SPRs) that are accessed explicitly using **mtspr** and **mfspr** instructions. These registers are listed in Table 2-1 in Section 2.2.1, “Special-Purpose Registers (SPRs).”
- e500-defined SPRs that are accessed explicitly using the **mtspr** and **mfspr** instructions. These registers are listed in Table 2-2 in Section 2.2.1, “Special-Purpose Registers (SPRs).”

Book E- and e500-defined SPRs are grouped by function as follows:

- Section 2.4, “Registers for Branch Operations.” This section includes descriptions of the count register (CTR) and the link register (LR).
- Section 2.5, “Processor Control Registers”
- Section 2.6, “Timer Registers”
- Section 2.7, “Interrupt Registers”
- Section 2.8, “Software-Use SPRs (SPRG0–SPRG7 and USPRG0)”
- Section 2.9, “Branch Target Buffer (BTB) Registers”
- Section 2.10, “Hardware Implementation-Dependent Registers”

- Section 2.11, “L1 Cache Configuration Registers”
- Section 2.12, “MMU Registers”
- Section 2.13, “Debug Registers”
- Section 2.14, “Signal Processing Engine (SPE) APU Registers”
- Section 2.16, “Instruction Address Resume Register (IARR)”

Book E provides 32-bit and 64-bit registers. All 32-bit registers are supported as defined in Book E. However, except for the 64-bit FPRs, which are not implemented in the e500, only bits 32–63 of Book E’s 64-bit registers (such as LR, CTR, the GPRs, SRR0, and CSRR0) are required to be implemented in hardware in a 32-bit Book E implementation. The e500 implements 64-bit GPRs, the top 32 bits of which are used only with the e500-specific signal processing unit (SPE APU) vector instructions.

Likewise, all Book E integer instructions defined to return a 64-bit result return only bits 32–63 of the result on a 32-bit Book E implementation. SPE APU vector instructions return 64-bit values; SPE APU scalar floating-point instructions return 32-bit values.

This chapter describes how the e500 implements registers defined by Book E. As with the instruction set and other aspects of the architecture, Book E defines some features very specifically, for example resources that ensure compatibility with implementations of the PowerPC ISA. However, because a principal goal of the Book E architecture is to offer flexibility among embedded processors and families of embedded processors, some resources are either defined as optional or are defined in a very general way, leaving specific details up to the implementation.

2.2.1 Special-Purpose Registers (SPRs)

SPRs are on-chip registers that are architecturally part of the processor core that control the use of the debug facilities, timers, interrupts, memory management unit, and other architected processor resources. They are accessed with the **mtspr** and **mfspir** instructions. Unlisted encodings are reserved for future use.

Table 2-1 summarizes SPRs defined in Book E. The SPR numbers are used in the instruction mnemonics. Bit 5 in an SPR number indicates whether an SPR is accessible from user- or supervisor software. An **mtspir** or **mfspir** instruction that specifies an unsupported SPR number is considered an invalid instruction. The e500 treats such invalid instructions as follows:

- If the invalid SPR is accessible in user mode, an illegal exception is taken.
- If supervisor software attempts to access an invalid supervisor-level SPR, results are undefined.
- If user software attempts to access an invalid supervisor-level SPR, a privilege exception is taken.

Table 2-1. Book E Special-Purpose Registers (by SPR Abbreviation)

SPR Abbreviation	Name	Defined SPR Number		Access	Supervisor Only	Section/ Page
		Decimal	Binary			
CSRR0	Critical save/restore register 0	58	00001 11010	Read/Write	Yes	2.7.1.1/2-23
CSRR1	Critical save/restore register 1	59	00001 11011	Read/Write	Yes	2.7.1.2/2-23
CTR	Count register	9	00000 01001	Read/Write	No	2.4.3/2-14
DAC1	Data address compare 1	316	01001 11100	Read/Write	Yes	2.13.4/2-56
DAC2	Data address compare 2	317	01001 11101			
DBCR0	Debug control register 0	308	01001 10100	Read/Write	Yes	2.13.1/2-50
DBCR1	Debug control register 1	309	01001 10101	Read/Write	Yes	
DBCR2	Debug control register 2	310	01001 10110	Read/Write	Yes	
DBSR	Debug status register	304	01001 10000	Read/Clear ¹	Yes	2.13.2/2-54
DEAR	Data exception address register	61	00001 11101	Read/Write	Yes	2.6.5/2-22
DEC	Decrementer	22	00000 10110	Read/Write	Yes	2.6.4/2-22
DECAR	Decrementer auto-reload	54	00001 10110	Write-only		
ESR	Exception syndrome register	62	00001 11110	Read/Write	Yes	2.7.1.8/2-26
IAC1	Instruction address compare 1	312	01001 11000	Read/Write	Yes	2.13.3/2-56
IAC2	Instruction address compare 2	313	01001 11001			

Table 2-1. Book E Special-Purpose Registers (by SPR Abbreviation) (continued)

SPR Abbreviation	Name	Defined SPR Number		Access	Supervisor Only	Section/ Page
		Decimal	Binary			
IVOR0	Critical input	400	01100 10000	Read/Write	Yes	2.7.1.7/2-25
IVOR1	Machine check interrupt offset	401	01100 10001			
IVOR2	Data storage interrupt offset	402	01100 10010			
IVOR3	Instruction storage interrupt offset	403	01100 10011			
IVOR4	External input interrupt offset	404	01100 10100			
IVOR5	Alignment interrupt offset	405	01100 10101			
IVOR6	Program interrupt offset	406	01100 10110			
IVOR7	Floating-point unavailable interrupt offset (not supported on the e500)	407	01100 10111			
IVOR8	System call interrupt offset	408	01100 11000			
IVOR9	Auxiliary processor unavailable interrupt offset (not supported on the e500)	409	01100 11001			
IVOR10	Decrementer interrupt offset	410	01100 11010			
IVOR11	Fixed-interval timer interrupt offset	411	01100 11011			
IVOR12	Watchdog timer interrupt offset	412	01100 11100			
IVOR13	Data TLB error interrupt offset	413	01100 11101			
IVOR14	Instruction TLB error interrupt offset	414	01100 11110			
IVOR15	Debug interrupt offset	415	01100 11111			
IVPR	Interrupt vector	63	00001 11111	Read/Write	Yes	2.7.1.6/2-25
LR	Link register	8	00000 01000	Read/Write	No	2.4.2/2-13
PID	Process ID register ²	48	00001 10000	Read/Write	Yes	2.12.1/2-41
PIR	Processor ID register	286	01000 11110	Read-only	Yes	2.5.2/2-16
PVR	Processor version register	287	01000 11111	Read-only	Yes	2.5.3/2-17

Table 2-1. Book E Special-Purpose Registers (by SPR Abbreviation) (continued)

SPR Abbreviation	Name	Defined SPR Number		Access	Supervisor Only	Section/ Page
		Decimal	Binary			
SPRG0	SPR general 0	272	01000 10000	Read/Write	Yes	2.8/2-31
SPRG1	SPR general 1	273	01000 10001	Read/Write	Yes	
SPRG2	SPR general 2	274	01000 10010	Read/Write	Yes	
SPRG3	SPR general 3	259	01000 00011	Read-only	No ³	
		275	01000 10011	Read/Write	Yes	
SPRG4	SPR general 4	260	01000 00100	Read-only	No	
		276	01000 10100	Read/Write	Yes	
SPRG5	SPR general 5	261	01000 00101	Read-only	No	
		277	01000 10101	Read/Write	Yes	
SPRG6	SPR general 6	262	01000 00110	Read-only	No	
		278	01000 10110	Read/Write	Yes	
SPRG7	SPR general 7	263	01000 00111	Read-only	No	
		279	01000 10111	Read/Write	Yes	
SRR0	Save/restore register 0	26	00000 11010	Read/Write	Yes	2.7.1.1/2-23
SRR1	Save/restore register 1	27	00000 11011	Read/Write	Yes	2.7.1.2/2-23
TBL	Time base lower	268	01000 01100	Read-only	No	2.6.3/2-21
		284	01000 11100	Write-only	Yes	
TBU	Time base upper	269	01000 01101	Read-only	No	
		285	01000 11101	Write-only	Yes	
TCR	Timer control register	340	01010 10100	Read/Write	Yes	2.6.1/2-19
TSR	Timer status register	336	01010 10000	Read/Clear ⁴	Yes	2.6.2/2-20
USPRG0	User SPR general 0 ⁵	256	01000 00000	Read/Write	No	2.8/2-31
XER	Integer exception register	1	00000 00001	Read/Write	No	2.3.2/2-9

¹ The DBSR is read using **mfsprr**. It cannot be directly written to. Instead, DBSR bits corresponding to 1 bits in the GPR can be cleared using **mtsprr**.

² Implementations may support more than one PID. The e500 implements the Book E–defined PID as PID0.

³ User-mode read access to SPRG3 is implementation-dependent.

⁴ The TSR is read using **mfsprr**. It cannot be directly written to. Instead, TSR bits corresponding to 1 bits in the GPR can be cleared using **mtsprr**.

⁵ USPRG0 is a separate physical register from SPRG0.

Table 2-2 describes the implementation-specific SPRs of the core complex. Compilers should recognize the mnemonic name given in Table 2-2 when parsing instructions.

Table 2-2. Implementation-Specific SPRs (by SPR Abbreviation)

SPR Abbreviation	Name	SPR Number	Access	Supervisor Only	Section/Page
BBEAR	Branch buffer entry address register	513	Read/Write	No	2.9.1/2-32
BBTAR	Branch buffer target address register	514	Read/Write	No	2.9.2/2-33
BUCSR	Branch unit control and status register	1013	Read/Write	Yes	2.9.3/2-33
HID0	Hardware implementation dependent reg 0	1008	Read/Write	Yes	2.10.1/2-35
HID1	Hardware implementation dependent reg 1	1009	Read/Write	Yes	2.10.1/2-35
IARR	Instruction address resume register	518	Read/Write	No	2.11.3/2-39
IVOR32	SPE APU unavailable interrupt offset	528	Read/Write	Yes	2.7.1.7/2-25
IVOR33	SPE floating-point data exception interrupt offset	529	Read/Write	Yes	
IVOR34	SPE floating-point round exception interrupt offset	530	Read/Write	Yes	
IVOR35	Performance monitor	531	Read/Write	Yes	
L1CFG0	L1 cache configuration register 0	515	Read-only	No	2.11.3/2-39
L1CFG1	L1 cache configuration register 1	516	Read-only	No	2.11.4/2-40
L1CSR0	L1 cache control and status register 0	1010	Read/Write	Yes	2.11.1/2-37
L1CSR1	L1 cache control and status register 1	1011	Read/Write	Yes	2.11.2/2-38
MAS0	MMU assist register 0	624	Read/Write	Yes	2.12.5.1/2-45
MAS1	MMU assist register 1	625	Read/Write	Yes	2.12.5.2/2-46
MAS2	MMU assist register 2	626	Read/Write	Yes	2.12.5.3/2-46
MAS3	MMU assist register 3	627	Read/Write	Yes	2.12.5.4/2-48
MAS4	MMU assist register 4	628	Read/Write	Yes	2.12.5.5/2-48
MAS5	MMU assist register 5. Defined by Motorola Book E implementation standard but not implemented in the e500.	629	Read/Write	Yes	—
MAS6	MMU assist register 6	630	Read/Write	Yes	2.12.5.6/2-49
MCAR	Machine check address register	573	Read-only	Yes	2.7.2.3/2-29
MCSR	Machine check syndrome register	572	Read/Write	Yes	2.7.2.4/2-29
MCSRR0	Machine-check save/restore register 0	570	Read/Write	Yes	2.7.2.1/2-28
MCSRR1	Machine-check save/restore register 1	571	Read/Write	Yes	2.7.2.2/2-28
MMUCFG	MMU configuration register	1015	Read-only	Yes	2.12.3/2-42
MMUCSR0	MMU control and status register 0	1012	Read/Write	Yes	2.12.2/2-41
NPIDR ¹	Nexus processor ID register	517	Read/Write	No	2.17/2-66

Table 2-2. Implementation-Specific SPRs (by SPR Abbreviation) (continued)

SPR Abbreviation	Name	SPR Number	Access	Supervisor Only	Section/Page
PID0	Process ID register 0. Book E defines only this PID register and refers to as PID rather than PID0.	48	Read/Write	Yes	2.12.1/2-41
PID1	Process ID register 1	633	Read/Write	Yes	
PID2	Process ID register 2	634	Read/Write	Yes	
SPEFSCR	Signal processing and embedded floating-point status and control register	512	Read/Write	No	2.14.1/2-57
TLB0CFG	TLB configuration register 0	688	Read-only	Yes	2.12.4/2-43
TLB1CFG	TLB configuration register 1	689	Read-only	Yes	2.12.4.2/2-44

¹ NPIDR is transferred to the NEXUS port whenever it is written.

2.3 Registers for Integer Operations

2.3.1 General Purpose Registers (GPRs)

Book E implementations provide 32 GPRs (GPR0–GPR31) for integer operations. The instruction formats provide 5-bit fields for specifying the GPRs to be used in the execution of the instruction. Each GPR is a 64-bit register and can be used to contain address and integer data, although all instructions except SPE APU vector instructions use and return 32-bit values in GPR bits 32–63.

2.3.2 Integer Exception Register (XER)

Bits in the integer exception register (XER) are set based on the operation of an instruction considered as a whole, not on intermediate results (for example, the Subtract from Carrying instruction (**subfc**), the result of which is specified as the sum of three values, sets bits in the XER based on the entire operation, not on an intermediate sum).

The XER contents are read into a GPR using **mf spr**. The contents of a GPR can be written to the XER using **mt spr**.

	32	33	34	35		56	57	63
Field	SO	OV	CA	—				Number of bytes
Reset	All zeros							
R/W	R/W							
SPR	SPR 1							

Figure 2-2. Integer Exception Register (XER)

Table 2-7 describes XER bit definitions.

Table 2-3. XER Field Description

Bits	Name	Description
32	SO	Summary overflow. Set when an instruction (except mtspr) sets the overflow bit. Once set, SO remains set until it is cleared by mtspr[XER] or mcrxr . SO is not altered by compare instructions or by other instructions (except mtspr[XER] and mcrxr) that cannot overflow. Executing mtspr[XER] , supplying the values 0 for SO and 1 for OV, causes SO to be cleared and OV to be set.
33	OV	Overflow. X-form add, subtract from, and negate instructions having OE=1 set OV if the carry out of bit 32 is not equal to the carry out of bit 33, and clear OV otherwise to indicate a signed overflow. X-form multiply low word and divide word instructions having OE=1 set OV if the result cannot be represented in 32 bits (mullw , divw , divwu) and clear OV otherwise. OV is not altered by compare instructions or by other instructions (except mtspr[XER] and mcrxr) that cannot overflow.
34	CA	Carry. Add carrying, subtract from carrying, add extended, and subtract from extended instructions set CA if there is a carry out of bit 32 and clear it otherwise. CA can be used to indicate unsigned overflow for add and subtract operations that set CA. Shift right algebraic word instructions set CA if any 1 bits are shifted out of a negative operand and clear CA otherwise. Compare instructions and instructions that cannot carry (except Shift Right Algebraic Word, mtspr[XER] , and mcrxr) do not affect CA.
35–56	—	Reserved, should be cleared.
57–63	No. of Bytes	Supports emulation of load and store string instructions. Specifies the number of bytes to be transferred by a load string indexed or store string indexed instruction.

2.4 Registers for Branch Operations

This section describes the registers that make up the branch and CR operations facilities of Book E.

2.4.1 Condition Register (CR)

The 32-bit condition register (CR) reflects the result of certain operations and provides a mechanism for testing and branching.

	32	35 36	39 40	43 44	47 48	51 52	55 56	59 60	63
Field	CR0	CR1	CR2	CR3	CR4	CR5	CR6	CR7	
Reset	All zeros								
R/W	R/W								

Figure 2-3. Condition Register (CR)

CR bits are grouped into eight 4-bit fields, CR0–CR7, which are set as follows:

- Specified CR fields can be set by a move to the CR from a GPR (**mtcrf**).
- A specified CR field can be set by a move to the CR from another CR field (**mcrf**), or from the XER (**mcrxr**).
- CR0 can be set as the implicit result of an integer instruction.

- A specified CR field can be set as the result of either an integer or a floating-point compare instruction.

Instructions are provided to perform logical operations on individual CR bits and to test individual CR bits (see Section 3.4.1.4.2, “Condition Register Logical Instructions”).

Note that Book E instructions, such as Branch Conditional (**bc**), CR logicals, and Move to Condition Register Field (**mtrcf**), that access CR bits determine the bit position by adding 32 to the value of the operand. For example, the BI operand in conditional branch instructions accesses the bit BI + 32, as shown in Table 2-4.

Table 2-4. BI Operand Settings for CR Fields

CRn Bits	CR Bits	BI	Description
CR0[0]	32	00000	Negative (LT)—Set when the result is negative. For SPE APU vector compare and vector test instructions: Set if the high-order element of rA is equal to the high-order element of rB; cleared otherwise.
CR0[1]	33	00001	Positive (GT)—Set when the result is positive (and not zero). For SPE APU vector compare and vector test instructions: Set if the low-order element of rA is equal to the low-order element of rB; cleared otherwise.
CR0[2]	34	00010	Zero (EQ)—Set when the result is zero. For SPE APU vector compare and vector test instructions: Set to the OR of the result of the compare of the high and low elements.
CR0[3]	35	00011	Summary overflow (SO). Copy of XER[SO] at the instruction's completion. For SPE APU vector compare and vector test instructions: Set to the AND of the result of the compare of the high and low elements.
CR1[0]	36	00100	Negative (LT) For SPE APU vector compare and vector test instructions: Set if the high-order element of rA is equal to the high-order element of rB; cleared otherwise.
CR1[1]	37	00101	Positive (GT) For SPE APU vector compare and vector test instructions: Set if the low-order element of rA is equal to the low-order element of rB; cleared otherwise.
CR1[2]	38	00110	Zero (EQ) For SPE APU vector compare and vector test instructions: Set to the OR of the result of the compare of the high and low elements.
CR1[3]	39	00111	Summary overflow (SO) For SPE APU vector compare and vector test instructions: Set to the AND of the result of the compare of the high and low elements.
CRn[0]	40 44 48 52 56 60	01000 01100 10000 10100 11000 11100	Less than (LT). For integer compare instructions: rA < SIMM or rB (signed comparison) or rA < UIMM or rB (unsigned comparison). For SPE APU vector compare and vector test instructions: Set if the high-order element of rA is equal to the high-order element of rB; cleared otherwise.
CRn[1]	41 45 49 53 57 61	01001 01101 10001 10101 11001 11101	Greater than (GT). For integer compare instructions: rA > SIMM or rB (signed comparison) or rA > UIMM or rB (unsigned comparison). For SPE APU vector compare and vector test instructions: Set if the low-order element of rA is equal to the low-order element of rB; cleared otherwise.

Table 2-4. BI Operand Settings for CR Fields (continued)

CRn Bits	CR Bits	BI	Description
CRn[2]	42	01010	Equal (EQ).
	46	01110	For integer compare instructions: rA = SIMM, UIMM, or rB.
	50	10010	For SPE APU vector compare and vector test instructions:
	54	10110	Set to the OR of the result of the compare of the high and low elements.
	58	11010	
	62	11110	
CRn[3]	43	01011	Summary overflow (SO).
	47	01111	For integer compare instructions, this is a copy of XER[SO] at the completion of the instruction.
	51	10011	
	55	10111	For SPE APU vector compare and vector test instructions:
	59	11011	Set to the AND of the result of the compare of the high and low elements.
	63	11111	

2.4.1.1 CR Setting for Integer Instructions

For all integer word instructions in which the Rc bit is defined and set, and for **addic.**, **andi.**, and **andis.**, the first three bits of CR0 (CR[32–34]) are set by signed comparison of bits 32–63 of the result to zero, and the fourth bit of CR0 (CR[35]) is copied from the final state of XER[SO]. The Rc bit is not defined for double-word integer operations.

```

if      (target_register)32-63 < 0 then c ← 0b100
else if (target_register)32-63 > 0 then c ← 0b010
else                                     c ← 0b001
CR0 ← c || XERSO

```

The value of any undefined portion of the result is undefined, the value placed into the first three bits of CR0 is undefined.

The bits of CR0 are interpreted as described in Table 2-5.

Table 2-5. CR0 Bit Descriptions

CR Bit	Name	Description
32	Negative (LT)	Bit 32 of the result is equal to 1.
33	Positive (GT)	Bit 32 of the result is equal to 0 and at least one bit 33–63 of the result is non-zero.
34	Zero (EQ)	Bits 32–63 of the result are equal to 0.
35	Summary overflow (SO)	This is a copy of the final state of XER[SO] at the completion of the instruction.

Note that CR0 may not reflect the true (infinitely precise) result if overflow occurs. See Section 3.4.1.1.1, “Integer Arithmetic Instructions.”

2.4.1.2 CR Setting for Store Conditional Instructions

CR0 is also set by the integer store conditional instruction, **stwcx.** See instruction descriptions in the EREF for detailed descriptions of how CR0 is set.

2.4.1.3 CR Setting for Compare Instructions

For compare instructions, a CR field specified by the BF field in the instruction is set to reflect the result of the comparison. The bits of CR field BF are interpreted as shown in Table 2-6. A complete description of how the bits are set is given in the instruction descriptions in Section 3.4.1.1.2, “Integer Compare Instructions.”

Table 2-6. CR Setting for Compare Instructions

CR Bit	Name	Description
4×BF + 32	LT	Less than - For signed-integer compare, GPR(RA) < SIMM or GPR(RB) - For unsigned-integer compare, GPR(RA) <u UIMM or GPR(RB)
4×BF + 33	GT	Greater than - For signed-integer Compare, GPR(RA) > SIMM or GPR(RB). - For unsigned-integer Compare, GPR(RA) >u UIMM or GPR(RB).
4×BF + 34	EQ	Equal For integer compare, GPR(RA) = SIMM, UIMM, or GPR(RB)..
4×BF + 35	SO	Summary overflow For integer compare, this is a copy of the final state of XER[SO] at the completion of the instruction.

2.4.2 Link Register (LR)

The link register (LR) can be used to provide the branch target address for the branch conditional to LR instruction, and it holds the return address after branch and link instructions.

	32	63
Field	Link address	
Reset	All zeros	
R/W	R/W	
SPR	SPR 8	

Figure 2-4. Link Register (LR)

The LR contents are read into a GPR using **mfspr**. The contents of a GPR can be written to the LR using **mtspr**. LR[62–63] are ignored by **bclr** registers.

Some processors may keep a stack of the LR values most recently set by branch and link instructions for obtaining the address of the next instruction. The e500 does not provide this facility.

2.4.3 Count Register (CTR)

CTR can be used to hold a loop count that can be decremented and tested during execution of branch instructions that contain an appropriately encoded BO field. If the CTR value is 0 before being decremented, it is -1 afterward. The entire CTR can be used to hold the branch target address for a Branch Conditional to CTR (**bcctrx**) instruction.

	32	63
Field	Count value	
Reset	All zeros	
R/W	R/W	
SPR	SPR 9	

Figure 2-5. Count Register (CTR)

The CTR contents are read into a GPR using **mfspr**. The contents of a GPR can be written to the CTR using **mtspr**.

2.5 Processor Control Registers

This section addresses machine state, processor ID, and processor version registers.

2.5.1 Machine State Register (MSR)

The machine state register (MSR) defines the state of the processor (that is, enabling and disabling of interrupts and debugging exceptions, enabling and disabling of address translation for instruction and data memory accesses, enabling and disabling some APUs, and specifying whether the processor is in supervisor or user mode).

MSR contents are automatically saved, altered, and restored by the interrupt-handling mechanism, as described in Section 5.6, “Interrupt Processing.” If a non-critical interrupt is taken, the MSR contents are automatically copied into SRR1. If a critical interrupt is taken, the MSR contents are automatically copied into CSRR1. When an **rfi** or **rfci** is executed, MSR contents are restored from SRR1 or CSRR1. The e500 implements the machine check interrupt differently than it is defined in Book E. When a machine check interrupt is taken, MCSRR0 and MCSRR1 hold the return address and MSR information. An implementation-specific Return from Machine Check Interrupt instruction, **rfmci**, is provided that restores MSR contents to MCSRR1 when it is executed.

The MSR contents are read into a GPR using **mfmsr**. The contents of a GPR can be written to the MSR using **mtmsr**. The write MSR external enable instructions (**wrttee** and **wrtteei**) can be used to set or clear MSR[EE] without affecting other MSR bits.

The following MSR bit assignments are unique to the e500 core complex.

	32	36	37	38	39	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
Field	—					WE	CE	—	EE	PR	FP	ME	FE0	UBLE	DE	FE1	—	—	IS	DS	—	PMM	—	—	
(e500)	—	UCLE	SPE	—	—	WE	CE	—	EE	PR	—	ME	—	UBLE	DE	—			IS	DS	—	PMM	—	—	
Reset	All zeros																								
R/W	R/W																								

Figure 2-6. Machine State Register (MSR)

Table 2-7 describes MSR fields.

Table 2-7. MSR Field Descriptions

Bits	Name	Description
32–36	—	Reserved, should be cleared. ¹
37	UCLE	User-mode cache lock enable. Used to restrict user-mode cache-line locking by the operating system 0 Any cache lock instruction executed in user-mode takes a cache-locking DSI exception and sets either ESR[DLK] or ESR[ILK]. This allows the operating system to manage and track the locking/unlocking of cache lines by user-mode tasks. 1 Cache-locking instructions can be executed in user-mode and they do not take a DSI for cache-locking (they may still take a DSI for access violations though).
38	SPE	SPE enable 0 If software attempts to execute an SPE APU instruction, the SPE APU unavailable exception is taken. 1 Software can execute any of the SPE APU instructions.
39–44	—	Reserved, should be cleared. ¹
45	WE	Wait state enable. Allows the core complex to signal a request for power management, according to the states of HID0[DOZE], HID0[NAP], and HID0[SLEEP]. 0 The processor is not in wait state and continues processing. No power management request is signalled to external logic. 1 The processor enters wait state by ceasing to execute instructions and entering low-power mode. Details of how wait state is entered and exited and how the processor behaves in the wait state are implementation-dependent. On the e500, MSR[WE] gates the DOZE, NAP, and SLEEP outputs from the core complex; as a result, these outputs negate to the external power management logic on entry to the interrupt and then return to their previous state on return from the interrupt. WE is cleared on entry to any interrupt and restored to its previous state upon return. See Chapter 6, “Power Management.”
46	CE	Critical enable 0 Critical input and watchdog timer interrupts are disabled. 1 Critical input and watchdog timer interrupts are enabled.
47	—	Preserved for Book III ILE
48	EE	External enable 0 External Input, decremter, and fixed-interval timer interrupts are disabled. 1 External input, decremter, and fixed-interval timer interrupts are enabled.
49	PR	User mode (problem state) 0 The processor is in supervisor mode, can execute any instruction, and can access any resource (for example, GPRs, SPRs, and the MSR). 1 The processor is in user mode, cannot execute any privileged instruction, and cannot access any privileged resource. PR also affects memory access control, as described in Section 11.2.4, “Permission Attributes.”

Table 2-7. MSR Field Descriptions (continued)

Bits	Name	Description
50	FP	Floating-point available. Book E defines the operation of FP as follows: 0 The processor cannot execute floating-point instructions, including floating-point loads, stores, and moves. Reserved and permanently cleared on the e500, indicating that it does not implement a Book E floating-point unit (FPU.) Setting it has no effect. 1 The processor can execute floating-point instructions.
51	ME	Machine check enable. 0 Machine check interrupts are disabled. 1 Machine check interrupts are enabled.
52	FE0	Floating-point exception mode 0. On the e500, reserved and permanently cleared, indicating the e500 does not implement a Book E floating-point unit. Setting it has no effect.
53	UBLE	Allocated for implementation-dependent use. In the e500, it is the user BTB lock enable bit. 0 Execution of the BTB lock instructions for user mode is disabled; privileged instruction exception taken instead. 1 Execution of the BTB lock instructions for user mode is enabled.
54	DE	Debug interrupt enable 0 Debug interrupts are disabled. 1 Debug interrupts are enabled if DBCR0[IDM] = 1. See the description of the DBSR[UDE] in Section 2.13.2, "Debug Status Register (DBSR)."
55	FE1	Floating-point exception mode 1. On the e500, reserved and permanently cleared, indicating the e500 does not implement a Book E floating-point unit. Setting it has no effect.
56	—	Reserved, should be cleared. ¹
57	—	Preserved for Book III IP
58	IS	Instruction address space 0 The processor directs all instruction fetches to address space 0 (TS=0 in the relevant TLB entry). 1 The processor directs all instruction fetches to address space 1 (TS=1 in the relevant TLB entry).
59	DS	Data address space 0 The processor directs data memory accesses to address space 0 (TS = 0 in the relevant TLB entry). 1 The processor directs data memory accesses to address space 1 (TS = 1 in the relevant TLB entry).
60	—	Reserved, should be cleared. ¹
61	PMM	Performance monitor mark bit. System software can set PMM when a marked process is running to enable statistics to be gathered only during the execution of the marked process. MSR[PR] and MSR[PMM] together define a state that the processor (supervisor or user) and the process (marked or unmarked) may be in at any time. If this state matches an individual state specified in the PMLCax, the state for which monitoring is enabled, counting is enabled. For more information, see Section 10.2, "Performance Monitor APU."
62–63	—	Preserved for Book III RI and LE, respectively.

¹ An MSR bit that is reserved may be altered by return from interrupt instructions.

2.5.2 Processor ID Register (PIR)

The processor ID register (PIR) contains a value that can be used to distinguish the processor from other processors in the system. The PIR value is read into a GPR using **mfspr**, as shown in Figure 2-7.

	32	47 48	63
Field	Version		Revision
Reset	All zeros		0000_0000_0000_(core_mid[0:3])
R/W	Read only		
SPR	SPR 286		

Figure 2-7. Processor ID Register (PIR)

2.5.3 Processor Version Register (PVR)

The processor version register (PVR) contains a value identifying the version and revision level of the processor. The PVR distinguishes between processors that differ in attributes that may affect software.

On the e500, the PVR is implemented as defined in Book E. The PVR contents are read into a GPR using **mfspvr**. The PVR is read-only, as shown in Figure 2-8.

	32	47 48	63
Field	Version		Revision
Reset	0x8020_0010		
R/W	Read only		
SPR	SPR 287		

Figure 2-8. Processor Version Register (PVR)

Table 2-8 describes PVR fields.

Table 2-8. PVR Field Descriptions

Bits	Name	Description
32–47	Version	A 16-bit number that identifies the version of the processor. Different version numbers indicate major differences between processors, such as which optional facilities and instructions are supported.
48–63	Revision	A 16-bit number that distinguishes between implementations of the version. Different revision numbers indicate minor differences between processors having the same version number, such as clock rate and engineering change level.

2.6 Timer Registers

The time base (TB), decremter (DEC), fixed-interval timer (FIT), and watchdog timer provide timing functions for the system. All of these must be initialized during start-up.

- The TB provides a long-period counter driven by an implementation-dependent frequency.
- The decremter, a counter that is updated at the same rate as the TB, provides a means of signaling an exception after a specified amount of time has elapsed unless:

Timer Registers

- DEC is altered by software in the interim, or
- The TB update frequency changes

The DEC is typically used as a general-purpose software timer.

- The fixed-interval timer is essentially a selected bit of the TB, which provides a means of signalling an exception whenever the selected bit transitions from 0 to 1, in a repetitive fashion. The fixed-interval timer is typically used to trigger periodic system maintenance functions. Software may select one of four bits in the TB to serve as the fixed-interval timer. Which bits may be selected is implementation-dependent.
- The watchdog timer is also a selected bit of the TB, which provides a means of signalling a critical class exception whenever the selected bit transitions from 0 to 1. In addition, if software does not respond in time to the initial exception (by clearing the associated status bits in the TSR before the next expiration of the watchdog timer interval), then a watchdog timer-generated processor reset may result, if so enabled. The watchdog timer is typically used to provide a system error recovery function.

The relationship of these timer facilities to each other is shown in Figure 2-9 and described in Chapter 9, “Timer Facilities.”

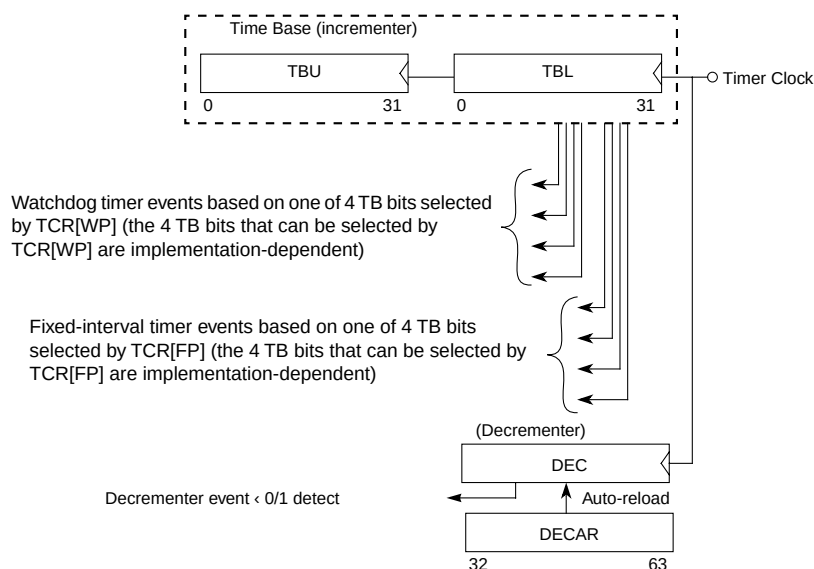


Figure 2-9. Relationship of Timer Facilities to TB

2.6.1 Timer Control Register (TCR)

The TCR, shown in Table 2-9, provides control information for the on-chip timer of the core complex. The core complex implements two fields not specified in Book E: TCR[WPEXT] and TCR[FPEXT].

The 32-bit timer control register (TCR), shown in Figure 2-10, controls decremter (see Section 2.6.4, “Decrementer Register”), fixed-interval timer (see Section 9.4, “Fixed-Interval Timer”), and watchdog timer (see Section 9.5, “Watchdog Timer”) options.

	32	33	34	35	36	37	38	39	40	41	42	43	46	47	50	51	63
Field	WP	WRC	WIE	DIE	FP	FIE	ARE	—	WPEXT	FPEXT	—						
Reset	All zeros																
R/W	R/W																
SPR	SPR 340																

Figure 2-10. Timer Control Register (TCR)

Table 2-9 describes the TCR fields.

Table 2-9. TCR Field Descriptions

Bits	Name	Description
32–33	WP	Watchdog timer period. When concatenated with WPEXT, specifies one of 64-bit locations of the time base used to signal a watchdog timer exception on a transition from 0 to 1. WPEXT[0–3],WP[0–1] = 0b00_0000 selects TBU[0] WPEXT[0–3],WP[0–1] = 0b11_1111 selects TBL[63]
34–35	WRC	Watchdog timer reset control. This value is written into TSR[WRS] when a watchdog event occurs. WRC may be set by software but cannot be cleared by software, except by a software-induced reset. Once written to a non-zero value, WRC may no longer be altered by software. 00 No watchdog timer reset will occur. 01 Force processor checkstop on second timeout of watchdog timer 10 Assert processor reset output (p_resetout_b) on second timeout of watchdog timer 11 Reserved
36	WIE	Watchdog timer interrupt enable 0 Watchdog timer interrupts disabled 1 Watchdog timer interrupts enabled
37	DIE	Decrementer interrupt enable 0 Decrementer interrupts disabled 1 Decrementer interrupts enabled
38–39	FP	Fixed interval timer period. When concatenated with FPEXT, FP specifies one of 64 bit locations of the time base used to signal a fixed-interval timer exception on a transition from 0 to 1. FPEXT[0–3],FP[0–1] = 0b00_0000 selects TBU[0] FPEXT[0–3],FP[0–1] = 0b11_1111 selects TBL[63]
40	FIE	Fixed interval interrupt enable 0 Fixed interval interrupts disabled 1 Fixed interval interrupts enabled

Table 2-9. TCR Field Descriptions (continued)

Bits	Name	Description
41	ARE	Auto-reload enable. Controls whether the value in DECAR is reloaded into the DEC when the DEC value reaches 0000_0001. 0 Auto-reload disabled 1 Auto-reload enabled
42	—	Reserved, should be cleared.
43–46	WPEXT	Watchdog timer period extension (see above description for WP).
47–50	FPEXT	Fixed-interval timer period extension (see above description for FP)
50–63	—	Reserved, should be cleared.

2.6.2 Timer Status Register (TSR)

As shown in Figure 2-11, the 32-bit TSR contains status on timer events and the most recent watchdog timer-initiated processor reset.

	32	33	34	35	36	37	38	63
Field	ENW	WIS	WRS	DIS	FIS	—		
Reset	All zeros							
R/W	Set by hardware. Read by using mfsp r and cleared by using mtsp r by writing ones to any TSR bit positions to be cleared and zeros in all other bit positions. The write-data to the TSR is not direct data, but a mask. A 1 causes the bit to be cleared, and a 0 has no effect.							
SPR	SPR 336							

Figure 2-11. Timer Status Register (TSR)

Table 2-10 describes TSR fields.

Table 2-10. TSR Field Descriptions

Bits	Name	Description
32	ENW	Enable next watchdog time. (See Section 9.5, “Watchdog Timer.”) 0 Action on next watchdog timer time-out is to set TSR[ENW] 1 Action on next watchdog timer time-out is governed by TSR[WIS] When a watchdog timer time-out occurs while WIS = 0 and the next watchdog time-out is enabled (ENW = 1), a watchdog timer exception is generated and logged by setting WIS. This is referred to as a watchdog timer first time out. A watchdog timer interrupt occurs if enabled by TCR[WIE] and MSR[CE]. To avoid another watchdog timer interrupt once MSR[CE] is reenabled (assuming TCR[WIE] is not cleared instead). The interrupt handler must reset TSR[WIS] by executing an mtsp r, setting WIS and any other bits to be cleared and a 0 in all other bits. The write-data to the TSR is not direct data, but is a mask. A 1 causes the bit to be cleared, and a 0 has no effect.
33	WIS	Watchdog timer interrupt status. (See Section 9.5, “Watchdog Timer.”) 0 A watchdog timer event has not occurred. 1 A watchdog timer event occurred. When MSR[CE]=1 and TCR[WIE]=1, a watchdog timer interrupt is taken. See the description of ENW for more information about how WIS is used.

Table 2-10. TSR Field Descriptions (continued)

Bits	Name	Description
34–35	WRS	Watchdog timer reset status. (See Section 9.5, “Watchdog Timer.”) Defined at reset (value = 00). Set to TCR[WRC] when a reset is caused by the watchdog timer.
36	DIS	Decrementer interrupt status. (See Section 9.3.3, “Decrementer Operation.”) 0 A decrementer event has not occurred. 1 A decrementer event occurred. When MSR[EE] = TCR[DIE] = 1, a decrementer interrupt is taken.
37	FIS	Fixed-interval timer interrupt status. (See Section 9.4, “Fixed-Interval Timer.”) 0 A fixed-interval timer event has not occurred. 1 A fixed-interval timer event occurred. When MSR[EE]=1 and TCR[FIE]=1, a fixed-interval timer interrupt is taken.
38–63	—	Reserved, should be cleared.

2.6.3 Time Base

The time base (TB), seen in Figure 2-12, is composed of two 32-bit registers, the time base upper (TBU) concatenated on the right with the time base lower (TBL). Operation of the time base facility is described in detail in Section 9.2.4, “Time Base.”

	32	63 32	63
Field	TBU		TBL
Reset	All zeros		All zeros
R/W	User read/Supervisor write		User read/Supervisor write
SPR	269 Read/285 Write		268 Read/284 Write

Figure 2-12. Timer Base Upper/Lower Registers (TBU/TBL)

The TB is interpreted as a 64-bit unsigned integer that is incremented periodically. Each increment adds 1 to the least-significant bit. The frequency at which the integer is updated is implementation-dependent.

The TB provides timing functions for the system. The TB is a volatile resource and must be initialized during start-up.

The TB provides a long-period counter driven by an implementation-dependent frequency.

TBU and TBL are accessed as follows:

- TBU contents are read using **mf spr**. The contents of a GPR can be written to the TBU using **mt spr**.
- TBL contents are read into a GPR using **mf spr**. The contents of GPR(RS) can be written to the TBL using **mt spr**.

TB is cleared at reset; otherwise, system software must initialize the TB.

TBL increments until its value becomes 0xFFFF_FFFF ($2^{32}-1$). At the next increment, its value becomes 0x0000_0000 and TBU is incremented. This process continues until the value in the TBU becomes 0xFFFF_FFFF and the value in the TBL becomes 0xFFFF_FFFF, or the value in the TB is interpreted as 0xFFFF_FFFF_FFFF_FFFF ($2^{64}-1$). At the next increment, the value in the TBU becomes 0x0000_0000 and the value in the TBL becomes 0x0000_0000. There is no interrupt or other indication when this occurs.

The period of the TB depends on the driving frequency. As an order of magnitude example, suppose that the TB is driven by a frequency of 100 MHz divided by 32. The period of the TB would be as follows:

$$T_{TB} = 2^{64} \times 32 / 100 \text{ MHz} = 5.90 \times 10^{12} \text{ seconds}$$

This value is approximately 187,000 years.

2.6.4 Decrementer Register

The contents of the 32-bit decrementer (DEC, Figure 2-13) are read into a GPR using **mfsprr**. The contents of a GPR can be written to the decrementer using **mtspr**.

	32	63
Field	Decrementer value	
Reset	All zeros	
R/W	R/W	
SPR	SPR 22	

Figure 2-13. Decrementer Register (DEC)

2.6.5 Decrementer Auto-Reload Register (DECAR)

The decrementer auto-reload register (DECAR), shown in Figure 2-14, supports the auto-reload feature of the decrementer. The contents of a GPR can be written to the DECAR using **mtspr**.

	32	63
Field	Decrementer auto-reload value	
Reset	All zeros	
R/W	Write only	
SPR	SPR 54	

Figure 2-14. Decrementer Auto-Reload Register (DECAR)

If the auto-reload function is enabled (TCR[ARE] = 1), the auto-reload value in DECAR is written to DEC when the DEC decrements from 0x0000_0001 to 0x0000_0000. Note that writing the DEC with zeros, by using an **mtspr[DEC]** does not automatically generate a

decrementer exception. Operation of the DECAR is described in Section 9.3.2, “Decrementer Auto-Reload Register (DECAR).”

2.7 Interrupt Registers

Section 2.7.1, “Interrupt Registers Defined by Book E,” and Section 2.7.2, “e500-Specific Interrupt Registers,” describe registers used for interrupt handling.

2.7.1 Interrupt Registers Defined by Book E

This section describes the following register bits and their fields:

- Section 2.7.1.1, “Save/Restore Register 0 (SRR0)”
- Section 2.7.1.2, “Save/Restore Register 1 (SRR1)”
- Section 2.7.1.3, “Critical Save/Restore Register 0 (CSRR0)”
- Section 2.7.1.5, “Data Exception Address Register (DEAR)”
- Section 2.7.1.6, “Interrupt Vector Prefix Register (IVPR)”
- Section 2.7.1.7, “Interrupt Vector Offset Registers (IVORs)”
- Section 2.7.1.8, “Exception Syndrome Register (ESR)”

2.7.1.1 Save/Restore Register 0 (SRR0)

On a noncritical interrupt, SRR0 holds the address of the instruction where the interrupted process should resume. The instruction is interrupt-specific, although for instruction-caused exceptions, it is typically the address of the instruction that caused the interrupt. See the individual interrupt descriptions in Section 5.7, “Interrupt Definitions.” When **rfi** executes, instruction execution continues at the address in SRR0, shown in Figure 2-15.

	32	63
Field	Next instruction address	
Reset	All zeros	
R/W	R/W	
SPR	SPR 26	

Figure 2-15. Save/Restore Register 0 (SRR0)

2.7.1.2 Save/Restore Register 1 (SRR1)

SRR1 (Figure 2-16) is provided to save and restore machine state on noncritical interrupts. When a noncritical interrupt is taken, MSR contents are placed in SRR1. When **rfi** executes, SRR1 contents are placed into MSR. SRR1 bits that correspond to reserved MSR bits are also reserved. These registers are not affected by **rfci** or **rfmci**.

Note that an MSR bit that is reserved may be altered by **rfi**, **rfci**, or **rfmci**

	32	63
Field	MSR state information	
Reset	All zeros	
R/W	R/W	
SPR	SPR 27	

Figure 2-16. Save/Restore Register 1 (SRR1)

2.7.1.3 Critical Save/Restore Register 0 (CSRR0)

CSRR0 is provided to save and restore machine state on critical interrupts. It is used by critical interrupts like SRR0 is used for standard interrupts: to hold the address of the instruction to which control is passed at the end of the interrupt handler. See the individual interrupt descriptions in Section 5.7, “Interrupt Definitions.” When **rfci** executes, instruction execution continues at the address in CSRR0, as is shown in Figure 2-17.

	32	63
Field	Next instruction address	
Reset	All zeros	
R/W	R/W	
SPR	SPR 58	

Figure 2-17. Critical Save/Restore Register 0 (CSRR0)

2.7.1.4 Critical Save/Restore Register 1 (CSRR1)

CSRR1 (Figure 2-18) is used to save and restore machine state on critical interrupts. When a critical interrupt is taken, MSR contents are placed into CSRR1. When **rfci** is executed, CSRR1 contents are restored into the MSR. CSRR1 bits that correspond to reserved MSR bits are also reserved. Reserved MSR bits may be altered by a return from interrupt instruction.

	32	63
Field	MSR state information	
Reset	All zeros	
R/W	R/W	
SPR	SPR 59	

Figure 2-18. Critical Save/Restore Register 1 (CSRR1)

2.7.1.5 Data Exception Address Register (DEAR)

DEAR is loaded with the effective address of a data access (caused by a load, store, or cache management instruction) that results in an alignment, data TLB miss, or DSI exception. Note that DEAR is read or written using **mfspr** and **mtspr**.

Figure 2-19 shows the format of DEAR.

	32	63
Field	Exception address	
Reset	All zeros	
R/W	R/W	
SPR	SPR 61	

Figure 2-19. Data Exception Address Register (DEAR)

2.7.1.6 Interrupt Vector Prefix Register (IVPR)

IVPR is used with IVORs to determine the vector address. IVPR[32–47] provides the high-order 16 bits of the address of the exception processing routines. The 16-bit vector offsets are concatenated to the right of IVPR[32–47] to form the address of the exception processing routine. Shown in Figure 2-20, IVPR[48–63] are reserved.

	32	47	48	63
Field	Interrupt vector offset		—	
Reset	All zeros			
R/W	R/W			
SPR	SPR 63			

Figure 2-20. Interrupt Vector Prefix Register (IVPR)

2.7.1.7 Interrupt Vector Offset Registers (IVORs)

IVORs hold the quad-word index from the base address provided by the IVPR for each interrupt type. IVOR0–IVOR15 are provided for defined interrupt types, as shown in Figure 2-21.

	32	47	48	59	60	63
Field	Interrupt vector offset					
(e500)	—		Interrupt vector offset		—	
Reset	All zeros					
R/W	R/W					
SPR	(See Table 2-11.)					

Figure 2-21. Interrupt Vector Offset Registers (IVOR)

SPR numbers corresponding to IVOR16–IVOR31 are reserved. IVOR32–IVOR47 and IVOR60–IVOR63 are reserved. SPR numbers for IVOR32–IVOR63 are allocated for implementation-dependent use (IVOR32–IVOR35 (SPR 528–531) are used by the e500). IVOR assignments are shown in Table 2-11.

Table 2-11. IVOR Assignments

IVOR Number	SPR	Interrupt Type
IVOR0	400	Critical input
IVOR1	401	Machine check
IVOR2	402	Data storage
IVOR3	403	Instruction storage
IVOR4	404	External input
IVOR5	405	Alignment
IVOR6	406	Program
IVOR7	407	Floating-point unavailable (not supported on the e500)
IVOR8	408	System call
IVOR9	409	Auxiliary processor unavailable (not supported on the e500)
IVOR10	410	Decrementer
IVOR11	411	Fixed-interval timer interrupt
IVOR12	412	Watchdog timer interrupt
IVOR13	413	Data TLB error
IVOR14	414	Instruction TLB error
IVOR15	415	Debug
IVOR16–IVOR31	—	Reserved for future architectural use
IVOR32	528	(e500-specific) SPE APU unavailable
IVOR33	529	(e500-specific) SPE floating-point data exception
IVOR34	530	(e500-specific) SPE floating-point round exception
IVOR35	531	(e500-specific) Performance monitor
IVOR36–IVOR63	—	Allocated for implementation-dependent use

2.7.1.8 Exception Syndrome Register (ESR)

ESR provides a syndrome to differentiate between different kinds of exceptions that can generate the same interrupt type. When such an interrupt is generated, bits corresponding to the specific exception that generated the interrupt are set and all other ESR bits are cleared. Other interrupt types do not affect ESR contents. The ESR does not need to be cleared by software. Table 2-12 shows ESR bit definitions. The e500 defines ESR[56] as the SPE exception bit (SPE). This bit is set whenever the processor takes an exception related to the execution of the SPE instructions. Note that the e500 does not use the ESR for

machine check interrupts, but instead uses the machine check syndrome register, MCSR, described in Section 2.7.2.4, “Machine Check Syndrome Register (MCSR).”

The exception syndrome register (ESR), shown in Figure 2-22, is defined in Book E but has additional bits defined for the e500.

	32	35	36	37	38	39	40	41	42	43	44	45	46	47	48	55	56	57	63
(e500)	—	PIL	PPR	PTR	—	ST	—	DLK	ILK	—	BO	—	—	—	—	SPE	—	—	—
Reset	All zeros																		
R/W	R/W																		
SPR	SPR 62																		

Figure 2-22. Exception Syndrome Register (ESR)

Table 2-12 describes ESR fields, showing the associated interrupts.

Note that an implementation may implement additional ESR bits to identify implementation-specific or architected interrupt types; the e500 implements ESR[ILK] and ESR[SPE].

NOTE

ESR information is incomplete, so system software may need to identify the type of instruction that caused the interrupt, examine the TLB entry, and examine the ESR to fully identify the exception or exceptions. For example, a data storage interrupt may be caused by both a protection violation exception and a byte-ordering exception. System software would have to look beyond ESR[BO], such as the state of MSR[PR] in SRR1 and the TLB entry page protection bits to determine if a protection violation also occurred.

Table 2-12 shows ESR bit definitions.

Table 2-12. Exception Syndrome Register (ESR) Definition

Bits	Name	Syndrome	Interrupt Types
32–35	—	Allocated	—
36	PIL	Illegal instruction exception	Program
37	PPR	Privileged instruction exception	Program
38	PTR	Trap exception	Program
39	—	Defined by Book E for floating-point operations (not supported on the e500). Reserved and permanently cleared because the e500 does not implement a Book E FPU. Setting it has no effect.	—
40	ST	Store operation	Alignment, data storage, data TLB error

Table 2-12. Exception Syndrome Register (ESR) Definition (continued)

Bits	Name	Syndrome	Interrupt Types
41	—	Reserved, should be cleared.	—
42	DLK	Cache locking. Settings are implementation-dependent. 0 Default 1 On the e500, DLK is set when a DSI occurs because dcbtls , dcbtstls , or dcblc is executed in user mode while MSR[UCLE] = 0.	Data storage
43	ILK	Set when a DSI occurs because icbtl or icblc is executed in user mode (MSR[PR] = 1) and MSR[UCLE] = 0	Data storage
45	—	Defined by Book E for unimplemented operation exception (not supported on the e500. Unimplemented instructions are handled as illegal instructions.)	—
46	BO	Byte-ordering exception	Data storage, instruction storage
47	—	Defined by Book E for imprecise exception (not supported on the e500)	Program
48–55	—	Reserved, should be cleared.	—
56	SPE	SPE exception bit (e500-specific). 0 Default 1 Any exception caused by an SPE instruction occurred.	SPE unavailable.
57–63	—	Allocated for implementation-dependent use	—

2.7.2 e500-Specific Interrupt Registers

This section describes machine check save/store and syndrome registers.

2.7.2.1 Machine Check Save/Restore Register 0 (MCSRR0)

When a machine check interrupt is taken, MCSRR0 (Figure 2-23) is set to the address of the instruction where the interrupted process should resume. The instruction is interrupt-specific, although it is typically the address of the instruction that caused the interrupt. See Section 5.7.2, “Machine Check Interrupt.” When **rfmci** is executed, instruction execution continues at the address in MCSRR0.

	32	63
Field	Next instruction address	
Reset	All zeros	
R/W	R/W	
SPR	SPR 570	

Figure 2-23. Machine Check Save/Restore Register 0 (MCSRR0)

2.7.2.2 Machine Check Save/Restore Register 1 (MCSRR1)

MCSRR1 is used to save and restore machine state on machine check interrupts. When a machine check interrupt is taken, MSR contents are placed into MCSRR1(Figure 2-24).

When **rfmci** is executed, MCSRR1 contents are restored to MSR. MCSRR1 bits that correspond to reserved MSR bits are also reserved. A reserved MSR bit may be altered by **rfmci**.

	32	63
Field	MSR state information	
Reset	All zeros	
R/W	R/W	
SPR	SPR 571	

Figure 2-24. Machine Check Save/Restore Register 1 (MCSRR1)

2.7.2.3 Machine Check Address Register (MCAR)

When the core complex takes a machine check interrupt, it updates MCAR (Figure 2-25) to indicate the address of the condition that caused the machine check. Note that if a machine check interrupt is caused by a signal, the contents of MCAR are not meaningful.

	32	63
Field	Machine check address	
Reset	All zeros	
R/W	Read only	
SPR	SPR 573	

Figure 2-25. Machine Check Address Register (MCAR)

2.7.2.4 Machine Check Syndrome Register (MCSR)

When the core complex takes a machine check interrupt, it updates MCSR to differentiate between machine check conditions. MCSR is shown in Figure 2-26.

Interrupt Registers

	32	33	34	35	36	37	38	39
Field	MCP	ICPERR	DCP_PERR	DCPERR	—	—	—	—
Reset	All zeros							
R/W	R/W							
	40	41	42	43	44	45	46	47
Field	—	—	—	—	—	—	—	—
Reset	All zeros							
R/W	R/W							
	48	49	50	51	52	53	54	55
Field	ABIST_IC	ABIST_DC	ABIST_ITLB	ABIST_DTLB	ABIST_L2TLB	ABIST_TLB1	ABIST_BTBT	—
Reset	All zeros							
R/W	R/W							
	56	57	58	59	60	61	62	63
Field	BUS_IAERR	BUS_RAERR	BUS_WAERR	BUS_IBERR	BUS_RBERR	BUS_WBERR	BUS_IPERR	BUS_RPERR
Reset	All zeros							
R/W	R/W							
SPR	SPR 572							

Figure 2-26. Machine Check Syndrome Register (MCSR)

Table 2-13 describes MCSR fields. The MCSR indicates whether a machine check condition is recoverable. When a condition bit is set, the core complex asserts MCP_OUT for system information. ABIST status is logged in MCSR[48–54]. These read-only bits do not initiate machine check (or any other exception). An ABIST bit being set indicates an error being detected in the corresponding module.

Table 2-13 shows MCSR bit definitions.

Table 2-13. Machine Check Syndrome Register (MCSR)

Bit	Name	Description	Recoverable
32	MCP	Machine check input pin	Maybe
33	ICPERR	Instruction cache parity error	Precise
34	DCP_PERR	Data cache push parity error	Maybe
35	DCPERR	Data cache parity error	Precise (cache-inhibited stwcx. instructions and guarded loads that initiate this error are deallocated before a machine check is taken. MCSRR0 holds the address of the next instruction, not the instruction that initiated the error.)
43–46	—	Reserved, should be cleared.	—

Table 2-13. Machine Check Syndrome Register (MCSR) (continued)

Bit	Name	Description	Recoverable
47	GL_CI	Set when a guarded load or cache-inhibited stwcx. provokes a parity error in the cache (ICPERR or DCPERR). In this case, the address saved in MCSRR0 is the address after the instruction that caused the error.	
48	ABIST_IC	Abist instruction cache error	NA
49	ABIST_DC	Abist data cache error	NA
50	ABIST_ITLB	Abist instruction TLB error	NA
51	ABIST_DTLB	Abist data TLB error	NA
52	ABIST_L2TLB	Abist L2 TLB error	NA
53	ABIST_TLB1	Abist TLB1 error	NA
54	ABIST_BTBT	Abist BTB error	NA
56	BUS_IAERR	Bus. Instruction address error	Maybe
57	BUS_RAERR	Bus. Read address error	Maybe
58	BUS_WAERR	Bus. Write address error	Maybe
59	BUS_IBERR	Bus. Instruction data bus error	Maybe
60	BUS_RBERR	Bus. Read data bus error	Maybe
61	BUS_WBERR	Bus. Write bus error	Maybe
62	BUS_IPERR	Bus—instruction parity error	Maybe
63	BUS_RPERR	Bus—read parity error	Maybe

2.8 Software-Use SPRs (SPRG0–SPRG7 and USPRG0)

Software-use SPRs (SPRG0–SPRG7 and USPRG0, shown in Figure 2-27) have no defined functionality. these are as follows:

- SPRG0–SPRG2—These registers can be accessed only in supervisor mode.
- SPRG3—This register can be written only in supervisor mode. It is readable in supervisor mode, but it is implementation-dependent whether it can be read in user mode. It is readable in user mode on the e500.
- SPRG4–SPRG7—These registers can be written only in supervisor mode. These registers are readable in supervisor or user mode.
- USPRG0—This register can be accessed in supervisor or user mode.

Field	32	Software-determined information				63
Reset		All zeros				
SPR R/W	SPRG0	272	Read/Write	Supervisor		
	SPRG1	273	Read/Write	Supervisor		
	SPRG2	274	Read/Write	Supervisor		
	SPRG3	259	Read-only	User/Supervisor		
		275	Read/Write	Supervisor		
	SPRG4	260	Read-only	User/Supervisor		
		276	Read/Write	Supervisor		
	SPRG5	261	Read-only	User/Supervisor		
		277	Read/Write	Supervisor		
	SPRG6	262	Read-only	User/Supervisor		
		278	Read/Write	Supervisor		
	SPRG7	263	Read-only	User/Supervisor		
		279	Read/Write	Supervisor		
	USPRG0	256	Read/Write	User/Supervisor		

Figure 2-27. Software-Use SPRs (SPRG0–SPRG7 and USPRG0)

Software-use SPRs are read into a GPR by using **mf spr** and are written by using **mt spr**.

2.9 Branch Target Buffer (BTB) Registers

This section describes the branch target buffer (BTB) registers and an MSR bit that affects the BTB. See Section 3.11.2, “Branch Target Buffer (BTB) Locking Instructions,” for more information about the BTB instructions and mechanism. Section 2.5.1, “Machine State Register (MSR),” has more information about MSR bits defined for the e500.

SPRs are defined in the core complex for enabling the locking and unlocking of entries in the BTB. These are called the branch buffer entry address register (BBEAR), the branch buffer target address register (BBTAR), and Branch unit control and status register (BUCSR). The user branch locking enable bit, MSR[UBLE], is defined to allow user mode programs to lock or unlock BTB entries.

2.9.1 Branch Buffer Entry Address Register (BBEAR)

BBEAR (Figure 2-28) be read and written in both the user and the privileged mode with the **mf spr** and **mt spr** instructions.

Field	32	Branch Buffer Entry Address		61 62	63
Reset		All zeros			
R/W		R/W			
SPR		SPR 513			

Figure 2-28. Branch Buffer Entry Address Register (BBEAR)

Table 2-15 describes BBEAR fields.

Table 2-14. BBEAR Field Descriptions

Bits	Name	Description
32–61	Branch buffer entry address	Branch buffer entry address bits 0–30
62–63	IAB[0–1]	Instruction after branch (with BBTAR[62]). 3-bit pointer that points to the instruction in the cache line after the branch. See the description in the bblels instruction in the EREF.

2.9.2 Branch Buffer Target Address Register (BBTAR)

BBTAR is read and written in both the user and the privileged mode with the **mf spr** and **mt spr** instructions. BBTAR is seen in Figure 2-29.

	32	61	62	63
Field	Branch buffer target address			IAB2 BDIRPR
Reset	All zeros			
R/W	R/W			
SPR	SPR 514			

Figure 2-29. Branch Buffer Target Address Register (BBTAR)

Table 2-15 describes BBTAR fields.

Table 2-15. BBTAR Field Descriptions

Bits	Name	Description
32–61	Branch buffer target address	Branch buffer target address bits 0–29
62	IAB2	Instruction after branch bit 2 (with BBEAR[62–63]). IAB is a 3-bit pointer that points to the instruction in the cache line after the branch. See the description in the bblels instruction in the EREF.
63	BDIRPR	Branch direction prediction. The user can pick the direction of the predicted branch. 0 The locked address is always predicted as not taken. 1 The locked address is always predicted as taken

2.9.3 Branch Unit Control and Status Register (BUCSR)

The BUCSR, shown in Figure 2-30, is used for general control and status of the branch target buffer (BTB).

Hardware Implementation-Dependent Registers

	32		53	54	55	56	57	58		62	63
Field	—			BBFI	BBLO	BBUL	BBLFC	—		BPEN	
Reset	All zeros										
R/W	R/W										
SPR	SPR 1013										

Figure 2-30. Branch Unit Control and Status Register (BUCSR)

BUCSR provides control of BTB locking, including the following:

- Enable or disable BTB locking
- Invalidate all BTB entries at once (flash invalidate)
- Unlock all BTB entries at once (flash lock clear)

Table 2-16 describes BUCSR fields.

Table 2-16. BUCSR Field Descriptions

Bits	Name	Description
32–53	—	Reserved, should be cleared.
54	BBFI	Branch buffer flash invalidate. Clearing and then setting BBFI flash clears the valid bit of all entries in the branch buffer; clearing occurs independently from the value of the enable bit (BPEN). BBFI is always read as 0.
55	BBLO	Branch buffer lock overflow status 0 Indicates a lock overflow condition was not encountered in the branch buffer 1 Indicates a lock overflow condition was encountered in the branch buffer This sticky bit is set by hardware and is cleared by writing 0 to this bit location.
56	BBUL	Branch buffer unable to lock 0 Indicates a lock overflow condition in the branch buffer 1 Indicates a lock set instruction failed in the branch buffer, for example, if the BTB is disabled. This sticky bit is set by hardware and is cleared by writing 0 to this bit location.
57	BBLFC	Branch buffer lock bits flash clear. Clearing and then setting BBLFC flash clears the lock bit of all entries in the branch buffer; clearing occurs independently from the value of the enable bit (BPEN). BBLFC is always read as 0.
58–62	—	Reserved, should be cleared.
63	BPEN	Branch prediction enable 0 Branch prediction disabled 1 Branch prediction enabled (enables BTB to predict branches)

2.10 Hardware Implementation-Dependent Registers

This section describes HID0 and HID1.

2.10.1 Hardware Implementation-Dependent Register (HID0)

The HID0 register, shown in Figure 2-31, is used for various configuration and control functions.

	32	33	39	40	41	42	43	46	47	48	49	50	51	62	63
Field	EMCP	—	DOZE	NAP	SLEEP	—	NHR	—	TBEN	SEL_TBCLK	—	—	—	NOPTI	
Reset	All zeros														
R/W	R/W														
SPR	SPR 1008														

Figure 2-31. HID0 Register

Table 2-17 describes HID0 fields.

Table 2-17. HID0 Field Descriptions

Bits	Name	Description
32	EMCP	Enable machine check pin, $\overline{MCP}$. Used to mask out further machine check exceptions caused by assertion of MCP. 0 $\overline{MCP}$ is disabled. 1 $\overline{MCP}$ is enabled. If $MSR[ME] = 0$, asserting $\overline{MCP}$ causes checkstop. If $MSR[ME] = 1$, asserting $\overline{MCP}$ causes a machine check exception.
33–39	—	Reserved, should be cleared.
40	DOZE	Doze power management mode. If $MSR[WE]$ is set, this bit controls DOZE mode. 0 Core not in doze mode 1 Core in doze mode
41	NAP	Nap power management mode. If $MSR[WE]$ is set, this bit controls NAP mode. 0 Core not in nap mode 1 Core in nap mode
42	SLEEP	Configure for sleep power management mode. Controls SLEEP mode if $MSR[WE]$ is set. 0 Core not in sleep mode 1 Core in sleep mode
43–48	—	Reserved, should be cleared.
49	TBEN	Time base enable 0 Time base disabled 1 Time base enabled
50	SEL_TBCLK	Select time base clock 0 Time base is based on the processor clock 1 Time base is based on TBCLK input
51–62	—	Reserved, should be cleared.
63	NOPTI	No-op the data and instruction cache touch instructions. 0 The dcbt , dcbtst , and icbt are enabled. 1 The dcbt , dcbtst , and icbt are treated as no-ops. Note that dcbtic and dcbtis are not treated as no-ops.

2.10.2 Hardware Implementation-Dependent Register 1 (HID1)

The HID1 register, shown in Figure 2-32, is used for bus configuration and control.

	32	37	38	46	47	48	49	50	51	52	53	54	55	56	57	59	60	63
Field	PLL_CFG	—		NEXEN	R1DPE	R2DPE	ASTME	ABE	—	MPXTT	MSHARS	SSHAR	ATS	—		MID		
Reset	All zeros																	
R/W	R/W																	
SPR	SPR 1009																	

Figure 2-32. HID1 Register

Table 2-18 describes HID1 fields.

Table 2-18. HID1 Field Descriptions

Bits	Name	Description
32–37	PLL_CFG	Reflected directly from the PLL_CFG input pins (read-only)
38–46	—	Reserved, should be cleared.
47	NEXEN	Nexus bus enable. Note that not all devices that implement the e500 core will support Nexus. 0 Nexus bus disabled 1 Nexus bus enabled
48	R1DPE	R1 data bus parity enable 0 R1 data bus parity checking disabled 1 R1 data bus parity checking enabled
49	R2DPE	R2 data bus parity enable 0 R2 data bus parity checking disabled 1 R2 data bus parity checking enabled
50	ASTME	Address bus streaming mode enable 0 Address bus streaming mode disabled 1 Address bus streaming mode enabled
51	ABE	Address broadcast enable 0 Address broadcasting disabled 1 Address broadcasting enabled
52	—	Reserved, should be cleared.
53	MPXTT	MPX re-map transfer type 0 TTx codes are not remapped 1 Certain TTx codes are remapped for MPX bus compatibility
54	MSHARS	MMU shared select 0 HID1[SSHAR] selects cache coherency model 1 MMU determines 3-or 4-state cache coherency model on a page-by-page basis.
55	SSHAR	Default cache coherency model 0 Three-state cache coherency model used 1 Four-state cache coherency model used
56	ATS	Atomic status (read-only) Indicates state of atomic status bit in bus unit.

Table 2-18. HID1 Field Descriptions (continued)

Bits	Name	Description
57–59	—	Reserved, should be cleared.
60–63	MID	Reflected directly from the MID input pins (read-only)

2.11 L1 Cache Configuration Registers

The Motorola Book E standards define registers that provide control and configuration and status information for the L1 cache implementation.

2.11.1 L1 Cache Control and Status Register 0 (L1CSR0)

The L1CSR0 register, defined as part of the Motorola Book E standards, is shown in Figure 2-33, is used for general control and status of the L1 data cache.

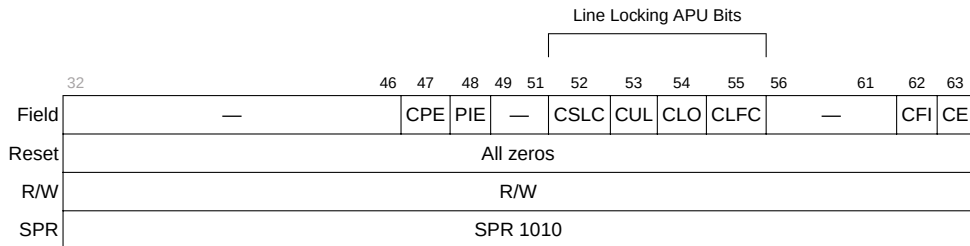


Figure 2-33. L1 Cache Control and Status Register 0 (L1CSR0)

Table 2-19 describes the L1CSR0 fields.

Table 2-19. L1CSR0 Field Descriptions

Bits	Name	Description
32–46	—	Reserved, should be cleared.
47	CPE	Cache parity enable 0 Parity checking of the cache disabled 1 Parity checking of the cache enabled
48	PIE	Parity injection enabled 0 Parity injection into cache disabled 1 Parity injection into cache enabled
49–51	—	Reserved, should be cleared.
52	CSLC	Cache snoop lock clear. Sticky bit set by hardware if a dcbi snoop (either internally or externally generated) invalidated a locked cache line. Note that the lock bit for that line is cleared whenever the line is invalidated. This bit can be cleared only by software. 0 The cache has not encountered an dcbi snoop that invalidated a locked line. 1 The cache has encountered an dcbi snoop that invalidated a locked line.

Table 2-19. L1CSR0 Field Descriptions (continued)

Bits	Name	Description
53	CUL	Cache unable to lock. Sticky bit set by hardware and cleared by writing 0 to this bit location. 0 Indicates a lock set instruction was effective in the cache 1 Indicates a lock set instruction was not effective in the cache
54	CLO	Cache lock overflow. Sticky bit set by hardware and cleared by writing 0 to this bit location. 0 Indicates a lock overflow condition was not encountered in the cache 1 Indicates a lock overflow condition was encountered in the cache
55	CLFC	Cache lock bits flash clear When written to a 1, hardware initiates a cache lock bits flash clear operation. Once the operation is complete, this bit is cleared. Writing a 1 during a flash clear operation causes an undefined operation. Writing a 0 during a flash clear operation is ignored. Clearing occurs regardless of the enable (CE) value.
56–61	—	Reserved, should be cleared.
62	CFI	Cache flash invalidate. Cache invalidate 0 No cache invalidate. Writing a 0 to CINV during an invalidation operation is ignored. 1 Cache invalidation operation. A cache invalidation operation is initiated by hardware. Once complete, this bit is cleared. Writing a 1 during an invalidation operation causes an undefined operation. Invalidation occurs regardless of the enable (CE) value.
63	CE	Cache enable 0 The cache is neither accessed or updated. 1 Enables cache operation.

2.11.2 L1 Cache Control and Status Register 1 (L1CSR1)

The L1CSR1 register, defined as part of the EIS, is shown in Figure 2-34. It is used for general control and status of the L1 instruction cache.

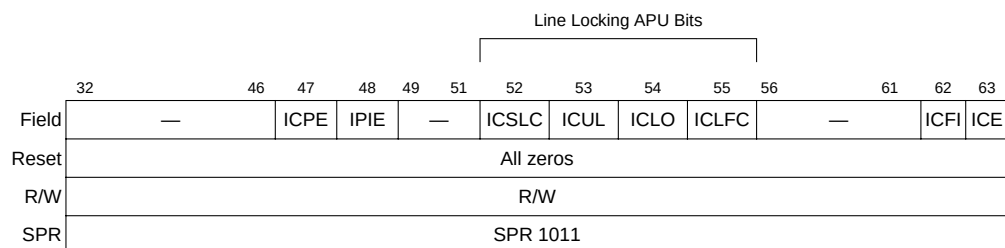
**Figure 2-34. L1 Cache Control and Status Register 1 (L1CSR1)**

Table 2-20 describes the L1CSR1 fields.

Table 2-20. L1CSR1 Field Descriptions

Bits	Name	Description
32–46	—	Reserved, should be cleared.
47	ICPE	Instruction cache parity enable 0 Parity checking of the instruction cache disabled 1 Parity checking of the instruction cache enabled
48	IPIE	Instruction parity injection enabled 0 Parity injection into instruction cache disabled 1 Parity injection into instruction cache enabled
49–51	—	Reserved, should be cleared.
52	ICSLC	Instruction cache snoop lock clear. Sticky bit set by hardware if an icbi snoop (either internally or externally generated) invalidated a locked line in the instruction cache. Note that the lock bit for that line is cleared whenever the line is invalidated. This bit can only be cleared by software. 0 The instruction cache has not encountered an icbi snoop that invalidated a locked line. 1 The instruction cache has encountered an icbi snoop that invalidated a locked line.
53	ICUL	Instruction cache unable to lock. Sticky bit set by hardware and cleared by writing 0 to this bit location. 0 Indicates a lock set instruction was effective in the instruction cache 1 Indicates a lock set instruction was not effective in the instruction cache
54	ICLO	Instruction cache lock overflow. Sticky bit set by hardware and cleared by writing 0 to this bit location. 0 Indicates a lock overflow condition was not encountered in the instruction cache 1 Indicates a lock overflow condition was encountered in the instruction cache
55	ICLFC	Instruction cache lock bits flash clear. Written to 0 and then 1 to flash clear the lock bit of all entries in the instruction cache; clearing occurs independently from the value of the enable bit (ICE).
56–61	—	Reserved, should be cleared.
62	ICFI	Instruction cache flash invalidate. Written to 0 and then 1 to flash clear the valid bit of all entries in the instruction cache; operates independently from the value of the enable bit (ICE).
63	ICE	Instruction cache enable 0 The instruction cache is neither accessed or updated. 1 Enables instruction cache operation.

2.11.3 L1 Cache Configuration Register 0 (L1CFG0)

The L1CFG0 register, defined as part of the EIS, is shown in Figure 2-35. It provides configuration information for the particular L1 data cache supplied with this version of the e500 core complex.

	32	33	34		38	39	40	41	42	43	44	45	49	50	52	53	55	56	63
Field	CARCH	—				CBSIZE	CREPL	CLA	CPA	—		CNWAY	—		CSIZE				
Reset	00	00_000				00	01	1	1	000_00		01_1	000		0000_1000				
R/W	Read only																		
SPR	SPR 515																		

Figure 2-35. L1 Cache Configuration Register 0 (L1CFG0)

Table 2-21 describes L1CFG0 fields.

Table 2-21. L1CFG0 Field Descriptions

Bits	Name	Description
32–33	CARCH	Cache architecture 00 Harvard 01 Unified
34–38	—	Reserved, should be cleared.
39–40	CBSIZE	Cache line size 0 32 bytes 1 64 bytes
41–42	CREPL	Cache replacement policy 0 True LRU 1 Pseudo LRU
43	CLA	Cache locking APU available 0 Unavailable 1 Available
44	CPA	Cache parity available 0 Unavailable 1 Available
45–49	—	Reserved, should be cleared.
50–52	CNWAY	Cache number of ways. 111Indicates 8 ways.
53–55	—	Reserved, should be cleared.
56–63	CSIZE	Cache size. 0x20: indicates 32 Kbytes (8 -> size in 4-Kbyte blocks)

2.11.4 L1 Cache Configuration Register 1 (L1CFG1)

The L1CFG1 register, shown in Figure 2-36, provides configuration information for the particular L1 instruction cache supplied with this version of the e500 core complex.

	32	38	39	40	41	42	43	44	45	52	53	63
Field	—	ICBSIZ	ICREPL	ICLA	ICPA	ICNWAY				ICSIZE		
Reset	0000_000	0_0	01	1	1	000_0011_1				000_0010_0000		
R/W	Read only											
SPR	SPR 516											

Figure 2-36. L1 Cache Configuration Register 1 (L1CFG1)

Table 2-22 describes the L1CFG1 fields.

Table 2-22. L1CFG1 Field Descriptions

Bits	Name	Description
32–38	—	Reserved, should be cleared.
39–40	ICBSIZ	Instruction cache block size. 00 Indicates block size of 32 bytes
41–42	ICREPL	Instruction cache replacement policy. 01Indicates pseudo-LRU policy

Table 2-22. L1CFG1 Field Descriptions (continued)

Bits	Name	Description
43	ICLA	Instruction cache locking available. 1 indicates available.
44	ICPA	Instruction cache parity available. 1 indicates available.
45–52	ICNWAY	Instruction cache number of ways. 011 Indicates eight ways.
53–63	ICSIZE	Instruction cache size. 0x8 indicates 32 Kbytes (8 -> size in 4-Kbyte blocks)

2.12 MMU Registers

This section describes the following MMU registers and their fields:

- Process ID registers (PID0–PID2)
- MMU control and status register 0 (MMUCSR0)
- MMU configuration register (MMUCFG)
- TLB configuration registers (TLBnCFG)
- MMU assist registers (MAS0–MAS4, MAS6)

2.12.1 Process ID Registers (PID0–PID2)

The Book E architecture defines that a process ID (PID) value be associated with each effective address (instruction or data) generated by the processor. Book E defines one PID register that maintains the value of the PID for the current process. The e500 implements two additional PID registers, PID1 and PID2. The number of PIDs implemented is indicated by the value of MMUCFG[NPIDSS]. PID values are used to construct virtual addresses for accessing memory. The e500 implements only PID[54–63] for the process ID, as shown in Figure 2-37.

	32	53	54	63
Field (Book E)	Process ID			
(e500)	—			Process ID
Reset	All zeros			
R/W	R/W			
SPR	SPR 48(PID0 ¹); SPR 633 (PID1 ²); SPR 634 (PID2 ²)			

¹ Book E defines PID0 and refers to it as PID.

² e500-specific registers

Figure 2-37. Process ID Register (PID0–PID2)

2.12.2 MMU Control and Status Register 0 (MMUCSR0)

The MMUCSR0 register (Figure 2-38) is used for general control of the L1 and L2 MMUs.

MMU Registers

	32	58	59	60	61	62	63
Field	—	IL1MMU_FI	DL1MMU_FI	L2TLB0_FI	L2TLB1_FI	—	
Reset	All zeros						
R/W	R/W						
SPR	SPR 1012						

Figure 2-38. MMU Control and Status Register 0 (MMUCSR0)

Table 2-23 describes the MMUCSR0 fields.

Table 2-23. MMUCSR0 Field Descriptions

Bits	Name	Description
32–58	—	Reserved, should be cleared.
59	IL1MMU_FI	Instruction L1 MMU (TLB) flash invalidate (write to 1 to invalidate)
60	DL1MMU_FI	Data L1 MMU (TLB) flash invalidate (write to 1 to invalidate)
61	L2TLB0_FI	TLB0 flash invalidate (write to 1 to invalidate)
62	L2TLB1_FI	TLB1 flash invalidate (write to 1 to invalidate)0No flash invalidate. Writing a 0 to this bit during an invalidation operation is ignored. 1 TLB1 invalidation operation. Hardware initiates a TLB1 invalidation operation. When this operation is complete, this bit is cleared. Writing a 1 during an invalidation operation is causes an undefined operation.
63	—	Reserved, should be cleared.

2.12.3 MMU Configuration Register (MMUCFG)

The read-only MMUCFG register, shown in Figure 2-39, provides configuration information about the e500 MMU.

	32	48	49	52	53	57	58	59	60	61	62	63
Field	—			NPIDS	PIDSIZE	—		NTLBS	MAVN			
Reset	0000_0000_0000_0000_0			001_1	001_11	00		01	00			
R/W	Read only											
SPR	SPR 1015											

Figure 2-39. MMU Configuration Register 1 (MMUCFG)

Table 2-24 describes the MMUCFG fields.

Table 2-24. MMUCFG Field Descriptions

Bits	Name	Description
32–48	—	Reserved, should be cleared.
49–52	NPIDS	Number of PID registers, a 4-bit field that indicates the number of PID registers provided by the processor. The e500 implements three PIDs.

Table 2-24. MMUCFG Field Descriptions (continued)

Bits	Name	Description
53–57	PIDSIZE	PID register size. The 5-bit value of PIDSIZE is one less than the number of bits in each of the PID registers implemented by the processor. The processor implements only the least significant PIDSIZE+1 bits in the PID registers. 00111: indicates 8-bit registers. This is the value presented by the e500
58–59	—	Reserved, should be cleared.
60–61	NTLBS	Number of TLBs. The value of NTLBS is one less than the number of software-accessible TLB structures that are implemented by the processor. NTLBS is set to one less than the number of TLB structures so that its value matches the maximum value of MAS0[TLBSEL].) 00 1 TLB 01 2 TLBs. This is the value presented by the e500. 10 3 TLBs 11 4 TLBs
62–63	MAVN	MMU architecture version number. Indicates the version number of the architecture of the MMU implemented by the processor. 0b00 indicates version 1.0.

2.12.4 TLB Configuration Registers (TLBnCFG)

The TLBnCFG read-only registers provide information about each specific TLB that is visible to the programming model.

2.12.4.1 TLB0 Configuration Register (TLB0CFG)

TLB0CFG, shown in Figure 2-40, provides configuration information for TLB0 of the L2 MMU supplied with this version of the e500 core complex.

	32	39 40	43 44	47 48	49	50 51 52	63
Field	ASSOC	MINSIZE	MAXSIZE	IPROT	AVAIL	—	NENTRY
e500 Reset Values	0000_0010	0001	0001	0	0	00	0001_0000_0000
R/W	Read only						
SPR	SPR 688						

Figure 2-40. TLB Configuration Register 0 (TLB0CFG)

Table 2-25 describes the TLB0CFG fields and shows the values for the e500.

Table 2-25. TLB0CFG Field Descriptions

Bits	Name	Description
32–39	ASSOC	Associativity of TLB0 0x02: indicates associativity is 2-way set associative
40–43	MINSIZE	Minimum page size of TLB0 0x1: indicates smallest page size is 4K
44–47	MAXSIZE	Maximum page size of TLB0 0x1: indicates maximum page size is 4k

Table 2-25. TLB0CFG Field Descriptions (continued)

Bits	Name	Description
48	IPROT	Invalidate protect capability of TLB0 0 Indicates invalidate protection capability not supported
49	AVAIL	Page size availability of TLB0 0 No variable-sized pages available (MINSIZE=MAXSIZE)
50–51	—	Reserved, should be cleared.
52–63	NENTRY	Number of entries in TLB0 0x100: TLB0 contains 256 entries

2.12.4.2 TLB1 Configuration Register 1 (TLB1CFG)

The TLB1CFG register, shown in Figure 2-41, provides configuration information for TLB1 of the L2 MMU supplied with this version of the e500 core complex.

	32	39	40	43	44	47	48	49	50	51	52	63
Field	ASSOC			MINSIZE		MAXSIZE		IPROT	AVAIL	—	NENTRY	
e500 reset values	0001_0000			0001		1001		1	1	00	0000_0001_0000	
R/W	Read only											
SPR	SPR 689											

Figure 2-41. TLB Configuration Register 1 (TLB1CFG)

Table 2-26 describes the TLB1CFG fields.

Table 2-26. TLB1CFG Field Descriptions

Bits	Name	Description
32–39	ASSOC	Associativity of TLB1 0x10: indicates associativity is 16
40–43	MINSIZE	Minimum page size of TLB1 0x1: indicates smallest page size is 4K
44–47	MAXSIZE	Maximum page size of TLB1 0x9: indicates maximum page size is 256 Mbyte
48	IPROT	Invalidate protect capability of TLB1 1 Indicates that TLB1 supports invalidate protection capability
49	AVAIL	Page size availability of TLB1 1 Indicates all page sizes between MINSIZE and MAXSIZE supported
50–51	—	Reserved, should be cleared.
52–63	NENTRY	Number of entries in TLB1 0x010: TLB1 contains 16 entries

2.12.5 MMU Assist Registers (MAS0–MAS4, MAS6)

MMU assist registers MAS0–MAS4, MAS6 are implementation-defined SPRs used by the MMU to manage pages and TLBs. They, along with MAS5, are defined by the Motorola implementation standard, as described in Section 11.3.5, “MMU Assist (MAS) Registers.” Note that some fields in these registers are re-defined in the e500.

2.12.5.1 MAS Register 0 (MAS0)

Figure 2-42 describes the format of MAS0. For the core complex, TLB0 is two-way set associative, so bits 45–51 of the effective address are used to index into TLB0 when it is accessed. ESEL then identifies which of the two indexed entries is to be referenced by the TLB operation (ESEL selects the way).

	32	34	35	36	43	44	47	48	62	63
Field	—	TLBSEL	—	—	ESEL	—	—	—	—	NV
Reset	All zeros									
R/W	R/W									
SPR	SPR 624									

Figure 2-42. MAS Register 0 (MAS0) Format

The MAS0 fields are described in Table 2-27.

Table 2-27. MAS0—MMU Read/Write and Replacement Control

Bit	Name	Comments or Function when Set
32–34	—	Reserved, should be cleared.
35	TLBSEL	Selects TLB for access. 0 TLB0 1 TLB1
36–43	—	Reserved, should be cleared.
44–47	ESEL	For the e500, ESEL serves as the way select for the corresponding TLB as follows: When TLBSEL = 00 (TLB0 selected), only bit 47 is used (and bits 44–46 should be cleared). This bit selects between way 0 and way 1 of TLB0. EA bits 45–51 are used to index into the TLB to further select the entry for the operation. When TLBSEL = 01 (TLB1 selected), all four bits are used to select one of 16 entries in the array.
48–62	—	Reserved, should be cleared.
63	NV	For the e500 implementation, this bit is the next victim bit value to be written to TLB0[NV] on execution of tlbwe . NV is also updated on TLB error exceptions (misses), tlbsx hit and miss cases as shown in Table 13-10, and on execution of tlbre . NV is updated with the calculated next victim bit for TLB0, based on the round-robin replacement algorithm. Note that this field is not defined for operations that specify TLB1 (when TLBSEL = 01).

2.12.5.2 MAS Register 1 (MAS1)

Figure 2-43 describes the format of MAS1. Note that while the Motorola Book E allows for a TID field of 12 bits, the TID field on the core complex is implemented as only 8 bits.

	32	33	34		39	40		47	48	50	51	52		55	56		63
Field	V	IPROT	—		TID			—		TS	TSIZE		—				
Reset	All zeros																
R/W	R/W																
SPR	SPR 625																

Figure 2-43. MAS Register 1 (MAS1) Format

The MAS1 fields are described in Table 2-28.

Table 2-28. MAS1 Field Descriptions—Descriptor Context and Configuration Control

Bits	Name	Descriptions
32	V	TLB valid bit. 0 This TLB entry is invalid. 1 This TLB entry is valid.
33	IPROT	Invalidate protect. Set to protect this TLB entry from invalidate operations due the execution of tlbivax (TLB1 only). Note that not all TLB arrays are necessarily protected from invalidation with IPROT. Arrays that support invalidate protection are denoted as such in the TLB configuration registers. 0 Entry is not protected from invalidation 1 Entry is protected from invalidation. See Section 13.3.2.1, “IPROT Invalidation Protection in TLB1.”
34–39	—	Reserved, should be cleared.
40–47	TID	Translation identity. 8-bit field that defines the process ID for this TLB entry. TID is compared with the current process IDs of the three effective address to be translated. A TID value of 0 defines an entry as global and matches with all process IDs.
48–50	—	Reserved, should be cleared.
51	TS	Translation space. This bit is compared with the IS or DS fields of the MSR (depending on the type of access) to determine if this TLB entry may be used for translation.
52–55	TSIZE	Translation size. This 4-bit field defines the page size of the TLB entry. For TLB arrays that contain fixed-size TLB entries, this field is ignored. For variable page size TLB arrays, the page size is 4^{TSIZE} Kbytes. Note that the Motorola Book E standard supports all 16 page sizes defined in Book E.
56–63	—	Reserved, should be cleared.

2.12.5.3 MAS Register 2 (MAS2)

Figure 2-44 shows the format of MAS2. Note that the core complex defines the SHAREN bit not defined by the Motorola Book E standard.

	32		51	52	53	54	55	56	57	58	59	60	61	62	63	
Field	EPN					—	SHAREN		—	X0	X1	W	I	M	G	E
Reset	All zeros															
R/W	R/W															
SPR	SPR 626															

Figure 2-44. MAS Register 2 (MAS2) Format

The MAS2 fields are described in Table 2-29.

Table 2-29. MAS2 Field Descriptions—EPN and Page Attributes

Bits	Name	Description
32–51	EPN	Effective page number. Depending on page size, only the bits associated with a page boundary are valid. Bits that represent offsets within a page are ignored and should be zero.
52–53	—	Reserved for implementation-specific use.
54	SHAREN	Enables cache fills to use shared cache state. 0 Cache fills do not use the shared cache state for this page. 1 Cache fills use the shared cache state for this page.
55–56	—	Reserved, should be cleared.
57	X0	Implementation-dependent page attribute
58	X1	Implementation-dependent page attribute
59	W	Write-through 0 This page is considered write-back with respect to the caches in the system. 1 All stores performed to this page are written through the caches to main memory.
60	I	Caching-inhibited 0 Accesses to this page are considered cacheable. 1 The page is considered caching-inhibited. All loads and stores to the page bypass the caches and are performed directly to main memory. A read or write to a caching-inhibited page affects only the memory element specified by the operation.
61	M	Memory coherence required 0 Memory coherence is not required. 1 Memory coherence is required. This allows loads and stores to this page to be coherent with loads and stores from other processors (and devices) in the system, assuming all such devices are participating in the coherence protocol.
62	G	Guarded 0 Accesses to this page are not guarded and can be performed before it is known if they are required by the sequential execution model. 1 All loads and stores to this page are performed without speculation (that is, they are known to be required).
63	E	Endianness. Determines endianness for the corresponding page. Little-endian operation is true little endian, which differs from the modified little-endian byte ordering model optionally available in previous devices that implement the PowerPC architecture. Section 11.1.1.5, “Endianness,” describes the Book E definition of endianness. 0 The page is accessed in big-endian byte order. 1 The page is accessed in true little-endian byte order.

2.12.5.4 MAS Register 3 (MAS3)

Figure 2-45 shows the format of MAS3. The core complex uses the same bit definitions as the Motorola Book E standard for MAS3 for 32-bit implementations.

	32		51	52	53	54		57	58	59	60	61	62	63
Field	RPN					—	U0–U3		UX	SX	UW	SW	UR	SR
Reset	All zeros													
R/W	R/W													
SPR	SPR 627													

Figure 2-45. MAS Register 3 (MAS3) Format

The MAS3 fields are described in Table 2-30.

Table 2-30. MAS3 Field Descriptions—RPN and Access Control

Bits	Name	Description
32–51	RPN	Real page number. Depending on page size, only the bits associated with a page boundary are valid. Bits that represent offsets within a page are ignored and should be zero.
52–53	—	Reserved, should be cleared.
54–57	U0–U3	User bits. These bits are associated with a TLB entry and can be used by system software. For example, these bits may be used to hold information useful to a page scanning algorithm or be used to mark more abstract page attributes.
58–63	PERMIS	Permission bits (UX, SX, UW, SW, UR, SR). User and supervisor read, write, and execute permission bits. See Table 11-2 for more information on the page permission bits as they are defined by Book E.

2.12.5.5 MAS Register 4 (MAS4)

Figure 2-46 shows the format of MAS4. Note that the core complex defines the SHAREND bit not defined by the Motorola Book E standard.

	32	33	34	35	36		43	44	45	46	47	48	50		51		52		55	56								63
Field	—	TLBSELD				—				TIDSELD				—	SHAREND		TSIZED		—	X0D	X1D	WD	ID	MD	GD	ED		
Reset	All zeros																											
R/W	R/W																											
SPR	SPR 628																											

Figure 2-46. MAS Register 4 (MAS4) Format

The MAS4 fields are described in Table 2-31.

Table 2-31. MAS4 Field Descriptions—Hardware Replacement Assist Configuration

Bits	Name	Description
32–33	—	Reserved, should be cleared.
34–35	TLBSEL	TLBSEL default value. 2-bit field that specifies the default value to be loaded in MAS0[TLBSEL] on a TLB miss exception. See Section 11.3.6.1, “TLB Miss Exception Types,” for more information.
36–45	—	Reserved, should be cleared.
46–47	TIDSELD	TID default selection value. 2-bit field that specifies which of the current PID registers should be used to load the MAS1[TID] field on a TLB miss exception.
		The e500 implementation defines this field as follows: 00 = PID0 01 = PID1 10 = PID2 11 = TIDZ (0x00) (all zeros)
48–50	—	Reserved, should be cleared.
51	SHAREN	Default SHAREN value
52–55	TSIZED	Default TSIZE value. Specifies the default value to be loaded into MAS1[TSIZE] on a TLB miss exception.
56	—	Reserved, should be cleared.
57	X0D	Default X0 value. Specifies the default value to be loaded into MAS2[X0] on a TLB miss exception.
58	X1D	Default X1 value. Specifies the default value to be loaded into MAS2[X1] on a TLB miss exception.
59	WD	Default W value. Specifies the default value to be loaded into MAS2[W] on a TLB miss exception.
60	ID	Default I value. Specifies the default value to be loaded into MAS2[I] on a TLB miss exception.
61	MD	Default M value. Specifies the default value to be loaded into MAS2[M] on a TLB miss exception.
62	GD	Default G value. Specifies the default value to be loaded into MAS2[G] on a TLB miss exception.
63	ED	Default E value. Specifies the default value to be loaded into MAS2[E] on a TLB miss exception.

2.12.5.6 MAS Register 6 (MAS6)

Figure 2-47 shows the format of MAS6. (MAS5 is not implemented in the e500 core complex.) Note that while the Motorola Book E allows for a SPIDx field of 12 bits, SPID0 on the core complex is only an 8-bit field.

	32	35	36	39	40	47	48	50	51	62	63		
Field	—				SPID0				—				SAS
Reset	All zeros												
R/W	R/W												
SPR	SPR 630												

Figure 2-47. MAS Register 6 (MAS6)

The MAS6 fields are described in Table 2-32.

Table 2-32. MAS 6 Field Descriptions

Bits	Name	Description
32–39	—	Reserved, should be cleared.
40–47	SPID0	Contains the 8-bit search PID0 value. Specifies the PID value (recent value of PID0) used when searching the TLB during execution of tlbsx .
48–62	—	Reserved, should be cleared.
63	SAS	Address space (AS) value for searches. Specifies the value of AS used when searching the TLB during execution of tlbsx .

2.13 Debug Registers

This section describes debug-related registers that are accessible to software running on the processor. These registers are intended for use by special debug tools and debug software, and not by general application or operating system code.

2.13.1 Debug Control Registers

The debug control registers, DBCR0–DBCR2, are used to enable debug events, reset the processor, control timer operation during debug events, and set the debug mode of the processor.

2.13.1.1 Debug Control Register 0 (DBCR0)

The contents of DBCR0 are read into a GPR using **mfsprr**; GPR contents can be written to DBCR0 using **mtspr**.

	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49		62	63
Field	—	IDM	RST	ICMP	BRT	IRPT	TRAP	IAC1	IAC2	—	—	DAC1	DAC2	RET	—	—	—	—	—	—	FT
Reset	All zeros																				
R/W	R/W																				
SPR	SPR 308																				

Figure 2-48. Debug Control Register 0 (DBCR0)

Table 2-33 provides bit definitions for DBCR0.

Table 2-33. DBCR0 Field Descriptions

Bits	Name	Description
32	—	Allocated for implementation-dependent use. The e500 does not implement this bit.
33	IDM	Internal debug mode. 0 Debug interrupts are disabled. No debug interrupts are taken and debug events are not logged. 1 If MSR[DE] = 1, the occurrence of a debug event or the recording of an earlier debug event in the DBSR when MSR[DE] = 0 or DBCR0[IDM] = 0 causes a debug interrupt. Programming note: Software must clear debug event status in the DBSR in the debug interrupt handler when a debug interrupt is taken before re-enabling interrupts via MSR[DE]. Otherwise, redundant debug interrupts are taken for the same debug event.
34–35	RST	Reset. Book E defines this field such that 00 is always no action and all other settings are implementation specific. The e500 implements these bits as follows: 0x Default (No action) 1x Causes a hard reset if MSR[DE] and DBCR0[IDM] are set. Always cleared on subsequent cycle.
36	ICMP	Instruction completion debug event enable 0 ICMP debug events are disabled 1 ICMP debug events are enabled Note: Instruction completion does not cause an ICMP debug event if MSR[DE]=0.
37	BRT	Branch taken debug event enable 0 BRT debug events are disabled 1 BRT debug events are enabled Note: Taken branches do not cause a BRT debug event if MSR[DE]=0.
38	IRPT	Interrupt taken debug event enable. This bit affects only noncritical interrupts. 0 IRPT debug events are disabled 1 IRPT debug events are enabled
39	TRAP	Trap debug event enable 0 TRAP debug events cannot occur 1 TRAP debug events can occur
40	IAC1	Instruction address compare 1 debug event enable 0 IAC1 debug events cannot occur 1 IAC1 debug events can occur
41	IAC2	Instruction address compare 2 debug event enable. 0 IAC2 debug events cannot occur 1 IAC2 debug events can occur
42	—	Not implemented on the E500; defined by Book E as instruction address compare 3 debug event enable.
43	—	Not implemented on the E500; defined by Book E as instruction address compare 4 debug event enable
44–45	DAC1	Data address compare 1 debug event enable 00 DAC1 debug events cannot occur 01 DAC1 debug events can occur only if a store-type data storage access 10 DAC1 debug events can occur only if a load-type data storage access 11 DAC1 debug events can occur on any data storage access

Table 2-33. DBCR0 Field Descriptions (continued)

Bits	Name	Description
46–47	DAC2	Data address compare 2 debug event enable 00 DAC2 debug events cannot occur 01 DAC2 debug events can occur only if a store-type data storage access 10 DAC2 debug events can occur only if a load-type data storage access 11 DAC2 debug events can occur on any data storage access
48	RET	Return debug event enable 0 RET debug events cannot occur 1 RET debug events can occur Note: An rfci does not cause an RET debug event if MSR[DE] = 0 at the time that rfci executes.
49–62	—	Reserved, should be cleared.
63	FT	Freeze timers on debug event 0 Enable clocking of timers 1 Disable clocking of timers if any DBSR bit is set (except MRR)

2.13.1.2 Debug Control Register 1

The contents of the DBCR1 are read using **mfsprr**. GPR contents can be written to DBCR1 using **mtspr**, as shown in Figure 2-46.

	32	33	34	35	36	37	38	39	40	41	42	63
Field	IAC1US	IAC1ER	IAC2US	IAC2ER	IAC12M	—						
Reset	All zeros											
R/W	R/W											
SPR	SPR 309											

Figure 2-49. Debug Control Register 1 (DBCR1)

Table 2-34 provides bit definitions for the DBCR1.

Table 2-34. DBCR1 Field Descriptions

Bits	Name	Description
32–33	IAC1US	Instruction address compare 1 user/supervisor mode 00 IAC1 debug events can occur 01 Reserved. 10 IAC1 debug events can occur only if MSR[PR]=0 11 IAC1 debug events can occur only if MSR[PR]=1
34–35	IAC1ER	Instruction address compare 1 effective/real mode 00 IAC1 debug events are based on effective addresses 01 Reserved on the e500. 10 IAC1 debug events are based on effective addresses and can occur only if MSR[IS]=0 11 IAC1 debug events are based on effective addresses and can occur only if MSR[IS]=1

Table 2-34. DBCR1 Field Descriptions (continued)

Bits	Name	Description
36–37	IAC2US	Instruction address compare 2 user/supervisor mode 00 IAC2 debug events can occur 01 Reserved 10 IAC2 debug events can occur only if MSR[PR]=0 11 IAC2 debug events can occur only if MSR[PR]=1
38–39	IAC2ER	Instruction address compare 2 effective/real mode 00 IAC2 debug events are based on effective addresses 01 Reserved on the e500. 10 IAC2 debug events are based on effective addresses and can occur only if MSR[IS]=0 11 IAC2 debug events are based on effective addresses and can occur only if MSR[IS]=1
40–41	IAC12M	Instruction address compare 1/2 mode 00 Exact address compare. IAC1 debug events can occur only if the address of the instruction fetch is equal to the value specified in IAC1. IAC2 debug events can occur only if the address of the instruction fetch is equal to the value specified in IAC2. 01 Address bit match. IAC1 and IAC2 debug events can occur only if the address of the instruction fetch, ANDed with the contents of IAC2 are equal to the contents of IAC1, plus ANDed with the contents of IAC2. If IAC1US≠IAC2US or IAC1ER≠IAC2ER, results are boundedly undefined. 10 Inclusive address range compare. IAC1 and IAC2 debug events can occur only if the address of the instruction fetch is greater than or equal to the value specified in IAC1 and less than the value specified in IAC2. If IAC1US≠IAC2US or IAC1ER≠IAC2ER, results are boundedly undefined. 11 Exclusive address range compare. IAC1 and IAC2 debug events can occur only if the address of the instruction fetch is less than the value specified in IAC1 or is greater than or equal to the value specified in IAC2. If IAC1US≠IAC2US or IAC1ER≠IAC2ER, results are boundedly undefined.
42–63	—	Reserved, should be cleared.

2.13.1.3 Debug Control Register 2 (DBCR2)

The contents of DBCR2 are read using **mfsprr**. GPR contents can be written to DBCR2 (Figure 2-47) using **mtsprr**.

	32	33	34	35	36	37	38	39	40	41	42	63
Field	DAC1US	DAC1ER	DAC2US	DAC2ER	DAC12M	—						
Reset	All zeros											
R/W	R/W											
SPR	SPR 310											

Figure 2-50. Debug Control Register 2 (DBCR2)

Table 2-35 provides bit definitions for DBCR2.

Table 2-35. DBCR2 Field Descriptions

Bits	Name	Description
32–33	DAC1US	Data address compare 1 user/supervisor mode 00 DAC1 debug events can occur 01 Reserved 10 DAC1 debug events can occur only if MSR[PR]=0 11 DAC1 debug events can occur only if MSR[PR]=1
34–35	DAC1ER	Data address compare 1 effective/real mode 00 DAC1 debug events are based on effective addresses 01 Reserved on the e500. 10 DAC1 debug events are based on effective addresses and can occur only if MSR[DS]=0 11 DAC1 debug events are based on effective addresses and can occur only if MSR[DS]=1
36–37	DAC2US	Data address compare 2 user/supervisor mode 00 DAC2 debug events can occur 01 Reserved 10 DAC2 debug events can occur only if MSR[PR]=0 11 DAC2 debug events can occur only if MSR[PR]=1
38–39	DAC2ER	Data address compare 2 effective/real mode 00 DAC2 debug events are based on effective addresses 01 Reserved on the e500. 10 DAC2 debug events are based on effective addresses and can occur only if MSR[DS]=0 11 DAC2 debug events are based on effective addresses and can occur only if MSR[DS]=1
40–41	DAC12M	Data address compare 1/2 mode 00 Exact address compare. DAC1 debug events can occur only if the address of the data storage access is equal to the value specified in DAC1. DAC2 debug events can occur only if the address of the data storage access is equal to the value specified in DAC2. 01 Address bit match. DAC1 and DAC2 debug events can occur only if the address of the data storage access, ANDed with the contents of DAC2 are equal to the contents of DAC1, also ANDed with the contents of DAC2. If DAC1US≠DAC2US or DAC1ER≠DAC2ER, results are boundedly undefined. 10 Inclusive address range compare. DAC1 and DAC2 debug events can occur only if the address of the data storage access is greater than or equal to the value specified in DAC1 and less than the value specified in DAC2. If DAC1US≠DAC2US or DAC1ER≠DAC2ER, results are boundedly undefined. 11 Exclusive address range compare. DAC1 and DAC2 debug events can occur only if the address of the data storage access is less than the value specified in DAC1 or is greater than or equal to the value specified in DAC2. If DAC1US≠DAC2US or DAC1ER≠DAC2ER, results are boundedly undefined.
42–63	—	Reserved, should be cleared.

2.13.2 Debug Status Register (DBSR)

Shown in Figure 2-51, the DBSR provides status information for debug events and for the most recent processor reset.

	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	63
Field	IDE	UDE	MRR		ICMP	BRT	IRPT	TRAP	IAC1	IAC2	—	—	DAC1R	DAC1W	DAC2R	DAC2W	RET	—	
Reset	00		Undefined		0000_0000_0000														
R/W	R/W																		
SPR	SPR 304																		

Figure 2-51. Debug Status Register (DBSR)

The DBSR is set through hardware but is read and cleared through software. DBSR is read using **mfsprr**. DBSR bits are cleared by writing ones to them; writing zeros has no effect. Table 2-36 describes DBSR bit definitions.

Table 2-36. DBSR Field Descriptions

Bits	Name	Description												
32	IDE	Imprecise debug event. Set if MSR[DE] = 0 and a debug event causes its respective DBSR bit to be set.												
33	UDE	Unconditional debug event. Set if an unconditional debug event occurred. See Section 8.4.8, "Unconditional Debug Event." If the UDE signal (level sensitive, active low) is asserted, DBSR[UDE] is affected as follows: <table> <tr> <td>MSR[DE]</td><td>DBCR0[IDM]</td><td>Action</td></tr> <tr> <td>X</td><td>0</td><td>No action.</td></tr> <tr> <td>0</td><td>1</td><td>DBSR[UDE] is set.</td></tr> <tr> <td>1</td><td>1</td><td>DBSR[UDE] is set and a debug interrupt is taken.</td></tr> </table>	MSR[DE]	DBCR0[IDM]	Action	X	0	No action.	0	1	DBSR[UDE] is set.	1	1	DBSR[UDE] is set and a debug interrupt is taken.
MSR[DE]	DBCR0[IDM]	Action												
X	0	No action.												
0	1	DBSR[UDE] is set.												
1	1	DBSR[UDE] is set and a debug interrupt is taken.												
34–35	MRR	Most recent reset. Set of three values when a reset occurs. Undefined at power-up. The e500 implements HRESET as follows: 0x No hard reset occurred since this bit was last cleared by software. 1x The previous reset was a hard reset.												
36	ICMP	Instruction complete debug event. Set if an instruction completion debug event occurred and DBCR0[ICMP] = 1. See Section 8.4.5, "Instruction Complete Debug Event."												
37	BRT	Branch taken debug event. Set if a branch taken debug event occurred (DBCR0[BRT]=1). See Section 8.4.4, "Branch Taken (BRT) Debug Event."												
38	IRPT	Interrupt taken debug event. Set if an interrupt taken debug event occurred (DBCR0[IRPT]=1). See Section 8.4.6, "Interrupt Taken (IRPT) Debug Event."												
39	TRAP	Trap instruction debug event. Set if a trap instruction debug event occurred (DBCR0[TRAP]=1). See Section 8.4.3, "Trap Debug Event."												
40	IAC1	Instruction address compare 1 debug event. Set if an IAC1 debug event occurred (DBCR0[IAC1]=1). See Section 8.4.1, "Instruction Address Compare Debug Event."												
41	IAC2	Instruction address compare 2 debug event. Set if an IAC2 debug event occurred (DBCR0[IAC2]=1). See Section 8.4.1, "Instruction Address Compare Debug Event."												
42–43	—	Reserved, should be cleared												
44	DAC1R	Data address compare 1 read debug event. Set if a read-type DAC1 debug event occurred (DBCR0[DAC1]=10 or 11). See Section 8.4.1, "Instruction Address Compare Debug Event."												
45	DAC1W	Data address compare 1 write debug event. Set if a write-type DAC1 debug event occurred (DBCR0[DAC1]=01 or 11). See Section 8.4.2, "Data Address Compare Debug Event."												
46	DAC2R	Data address compare 2 read debug event. Set if a read-type DAC2 debug event occurred (DBCR0[DAC2]=10 or 11). See Section 8.4.2, "Data Address Compare Debug Event."												

Table 2-36. DBSR Field Descriptions (continued)

Bits	Name	Description
47	DAC2W	Data address compare 2 write debug event. Set if a write-type DAC2 debug event occurred (DBCR0[DAC2] =01 or 11). See Section 8.4.2, “Data Address Compare Debug Event.”
48	RET	Return debug event. Set if a return debug event occurred (DBCR0[RET]=1). See Section 8.4.7, “Return Debug Event.”
49–63	—	Reserved, should be cleared.

2.13.3 Instruction Address Compare Registers (IAC1–IAC4)

The instruction address compare registers (IAC1–IAC4) are each 64 bits, with bits 62–63 being reserved, as shown in Figure 2-52. The e500 does not implement IAC3 and IAC4.

	32	61 62 63
Field	Instruction address	—
Reset	All zeros	
R/W	R/W	
SPR	SPR 312 (IAC1);SPR 313 (IAC2);SPR 314 (IAC3);SPR 315 (IAC4)	

Figure 2-52. Instruction Address Compare Registers (IAC1–IAC4)

A debug event may be enabled to occur upon an attempt to execute an instruction from an address specified in one of the IAC registers, inside or outside a range specified by IAC1 and IAC2, or to blocks of addresses specified by the combination of the IAC1 and IAC2. Because all instruction addresses are required to be word-aligned, the two low-order bits of the IAC registers are reserved and do not participate in the comparison to the instruction address. See Section 8.4.1, “Instruction Address Compare Debug Event.”

The contents of an IAC register is read into a GPR using **mf spr**. The contents of a GPR can be written to an IAC using **mt spr**.

2.13.4 Data Address Compare Registers (DAC1–DAC2)

The data address compare 1 register (DAC1) and data address compare 2 register (DAC2), shown in Figure 2-53, are each 64 bits. A debug event may be enabled to occur upon loads, stores, or cache operations to an address specified in either DAC1 or DAC2, inside or outside a range specified by the DAC1 and DAC2, or to blocks of addresses specified by the combination of the DAC1 and DAC2. See Section 8.4.2, “Data Address Compare Debug Event.”

Field	32	63
Reset	All zeros	
R/W	R/W	
SPR	SPR 316 (DAC1); SPR 317 (DAC2)	

Figure 2-53. Data Address Compare Registers (DAC1–DAC2)

The contents of DAC1 or DAC2 are read into a GPR using **mfspr**. The contents of a GPR can be written to DAC1 or DAC2 using **mtspr**.

The contents of DAC1 or DAC2 are compared to the address generated by a data storage access instruction.

2.14 Signal Processing Engine (SPE) APU Registers

The SPE includes the following two registers:

- The signal processing and embedded floating-point status and control register (SPEFSCR), described in Section 2.14.1, “Signal Processing and Embedded Floating-Point Status and Control Register (SPEFSCR).”
- A 64-bit accumulator, described in Section 2.14.2, “Accumulator (ACC).”

2.14.1 Signal Processing and Embedded Floating-Point Status and Control Register (SPEFSCR)

The SPEFSCR, shown in Figure 2-54, is used for status and control of SPE instructions.

	High-Word Error Bits										Status Bits						
	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	
Field	SOVH	OVH	FGH	FXH	FINVH	FDBZH	FUNFH	FOVFH	—	FINXS	FINVS	FDBZS	FUNFS	FOVFS	MODE		
Reset	0000_0000_0000_0000																
R/W	R/W																

	Enable Bits															
	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
Field	SOV	OV	FG	FX	FINV	FDBZ	FUNF	FOVF	—	FINXE	FINVE	FDBZE	FUNFE	FOVFE	FRMC	
Reset	0000_0000_0000_0000															
R/W	R/W															
SPR	SPR 512															

Figure 2-54. Signal Processing and Embedded Floating-Point Status and Control Register (SPEFSCR)

Table 2-37 describes SPEFSCR bits.

Table 2-37. SPEFSCR Field Descriptions

Bits	Name	Function
32	SOVH	Summary integer overflow high. Set whenever an instruction (except mtspr) sets OVH. SOVH remains set until it is cleared by an mtspr[SPEFSCR] .
33	OVH	Integer overflow high. An overflow occurred in the upper half of the register while executing a SPE integer instruction
34	FGH	Embedded floating-point guard bit high. Floating-point guard bit from the upper half. The value is undefined if the processor takes a floating-point exception due to input error, floating-point overflow, or floating-point underflow.
35	FXH	Embedded floating-point sticky bit high. Floating bit from the upper half. The value is undefined if the processor takes a floating-point exception due to input error, floating-point overflow or floating-point underflow.
36	FINVH	Embedded floating-point invalid operation error high. Set when an input value on the high side is a NaN, Inf, or Denorm. Also set on a divide if both the dividend and divisor are zero.
37	FDBZH	Embedded floating-point divide by zero error high. Set if the dividend is non-zero and the divisor is zero.
38	FUNFH	Embedded floating-point underflow error high
39	FOVFH	Embedded floating-point overflow error high
40–41	—	Reserved, should be cleared.
42	FINXS	Embedded floating-point inexact sticky. $FINXS = FINXS \mid FGH \mid FXH \mid FG \mid FX$
43	FINVS	Embedded floating-point invalid operation sticky. Location for software to use when implementing true IEEE floating point.
44	FDBZS	Embedded floating-point divide by zero sticky. $FDBZS = FDBZS \mid FDBZH \mid FDBZ$
45	FUNFS	Embedded floating-point underflow sticky. Storage location for software to use when implementing true IEEE floating point.
46	FOVFS	Embedded floating-point overflow sticky. Storage location for software to use when implementing true IEEE floating point.
47	MODE	Embedded floating-point mode (read-only on e500)
48	SOV	Integer summary overflow. Set whenever an SPE instruction (except mtspr) sets OV. SOV remains set until it is cleared by mtspr[SPEFSCR] .
49	OV	Integer overflow. An overflow occurred in the lower half of the register while a SPE integer instruction is being executed.
50	FG	Embedded floating-point guard bit. Floating-point guard bit from the lower half. The value is undefined if the processor takes a floating-point exception due to input error, floating-point overflow, or floating-point underflow.
51	FX	Embedded floating-point sticky bit. Floating bit from the lower half. The value is undefined if the processor takes a floating-point exception due to input error, floating-point overflow or floating-point underflow.
52	FINV	Embedded floating-point invalid operation error. Set when an input value on the high side is a NaN, Inf, or Denorm. Also set on a divide if both the dividend and divisor are zero.
53	FDBZ	Embedded floating-point divide by zero error. Set of the dividend is non-zero and the divisor is zero.

Table 2-37. SPEFSCR Field Descriptions (continued)

Bits	Name	Function
54	FUNF	Embedded floating-point underflow error
55	FOVF	Embedded floating-point overflow error
56	—	Reserved, should be cleared
57	FINXE	Embedded floating-point inexact enable
58	FINVE	Embedded floating-point invalid operation/input error exception enable 0 Exception disabled 1 Exception enabled If the exception is enabled, a floating-point data exception is taken if FINV or FINVH is set by a floating-point instruction.
59	FDBZE	Embedded floating-point divide-by-zero exception enable 0 Exception disabled 1 Exception enabled If the exception is enabled, a floating-point data exception is taken if FDBZ or FDBZH is set by a floating-point instruction.
60	FUNFE	Embedded floating-point underflow exception enable 0 Exception disabled 1 Exception enabled If the exception is enabled, a floating-point data exception is taken if FUNF or FUNFH is set by a floating-point instruction.
61	FOVFE	Embedded floating-point overflow exception enable 0 Exception disabled 1 Exception enabled If the exception is enabled, a floating-point data exception is taken if FOVF or FOVFH is set by a floating-point instruction.
62–63	FRMC	Embedded floating-point rounding mode control 00 Round to nearest 01 Round toward zero 10 Round toward +infinity 11 Round toward -infinity

2.14.2 Accumulator (ACC)

The 64-bit architectural accumulator register holds the results of the multiply accumulate (MAC) forms of SPE integer instructions. The accumulator allows back-to-back execution of dependent MAC instructions, something that is found in the inner loops of DSP code such as FIR filters. The accumulator is partially visible to the programmer in that its results do not have to be explicitly read to use them. Instead, they are always copied into a 64-bit destination GPR specified as part of the instruction. The accumulator however has to be explicitly cleared when starting a new MAC loop. Based upon the type of instruction, an accumulator can hold either a single 64-bit value or a vector of two 32-bit elements.

The Initialize Accumulator instruction (**evmra**) is provided to initialize the accumulator. This instruction is described in Chapter 3, “Instruction Set.”

The accumulator is described in detail in Section 10.3.1.1.3, “Accumulator.”

2.15 Performance Monitor Registers (PMRs)

The Motorola Book E implementation standards defines a set of register resources used exclusively by the performance monitor. PMRs are similar to the SPRs defined in the Book E architecture and are accessed by **mtpmr** and **mfpmr** instructions, which are also defined by the EIS. Table 2-38 lists supervisor-level PMRs. User-level software that attempts to read or write supervisor-level PMRs causes a privilege exception. For more information, see Section 10.2, “Performance Monitor APU.”

Table 2-38. Performance Monitor Registers—Supervisor Level

Abbreviation	Register Name	PMR Number	pmr[0–4]	pmr[5–9]	Section/ Page
PMC0	Performance monitor counter 0	16	00000	10000	2.15.7/2-64
PMC1	Performance monitor counter 1	17	00000	10001	
PMC2	Performance monitor counter 2	18	00000	10010	
PMC3	Performance monitor counter 3	19	00000	10011	
PMGC0	Performance monitor global control register 0	400	01100	10000	2.15.1/2-61
PMLCa0	Performance monitor local control a0	144	00100	10000	2.15.3/2-62
PMLCa1	Performance monitor local control a1	145	00100	10001	
PMLCa2	Performance monitor local control a2	146	00100	10010	
PMLCa3	Performance monitor local control a3	147	00100	10011	
PMLCb0	Performance monitor local control b0	272	01000	10000	2.15.5/2-63
PMLCb1	Performance monitor local control b1	273	01000	10001	
PMLCb2	Performance monitor local control b2	274	01000	10010	
PMLCb3	Performance monitor local control b3	275	01000	10011	

User-level PMRs in Table 2-39 are read-only and are accessed with **mfpmr**. Attempting to write user-level registers in supervisor or user mode causes an illegal instruction exception.

Table 2-39. Performance Monitor Registers—User Level (Read-Only)

Abbreviation	Register Name	PMR Number	pmr[0–4]	pmr[5–9]	Section/ Page
UPMC0	User performance monitor counter 0	0	00000	00000	2.15.8/2-65
UPMC1	User performance monitor counter 1	1	00000	00001	
UPMC2	User performance monitor counter 2	2	00000	00010	
UPMC3	User performance monitor counter 3	3	00000	00011	

Table 2-39. Performance Monitor Registers—User Level (Read-Only) (continued)

Abbreviation	Register Name	PMR Number	pmr[0–4]	pmr[5–9]	Section/ Page
UPMLCa0	User performance monitor local control a0	128	00100	00000	2.15.4/2-63
UPMLCa1	User performance monitor local control a1	129	00100	00001	
UPMLCa2	User performance monitor local control a2	130	00100	00010	
UPMLCa3	User performance monitor local control a3	131	00100	00011	
UPMLCb0	User performance monitor local control b0	256	01000	00000	2.15.6/2-64
UPMLCb1	User performance monitor local control b1	257	01000	00001	
UPMLCb2	User performance monitor local control b2	258	01000	00010	
UPMLCb3	User performance monitor local control b3	259	01000	00011	
UPMGC0	User performance monitor global control register 0	384	01100	00000	2.15.2/2-62

2.15.1 Global Control Register 0 (PMGC0)

The performance monitor global control register (PMGC0), shown in Figure 2-55, controls all performance monitor counters.

	32	33	34	35	63
Field	FAC	PMIE	FCECE	—	
Reset	All zeros				
R/W	PMGC0: Supervisor R/W; UPMGC0: User read-only				
PMR	PMGC0: PMR400; UPMGC0: PMR384				

**Figure 2-55. Performance Monitor Global Control Register 0 (PMGC0)
User Performance Monitor Global Control Register 0 (UPMGC0)**

PMGC0 is cleared by a hard reset. Reading this register does not change its contents. Table 2-40 describes PMGC0 fields.

Table 2-40. PMGC0 Field Descriptions

Bits	Name	Description
32	FAC	Freeze all counters. When FAC is set by hardware or software, PMLCx[FC] maintains its current value until it is changed by software. 0 The PMCs are incremented (if permitted by other PM control bits). 1 The PMCs are not incremented.
33	PMIE	Performance monitor interrupt enable. Enabled conditions and events are described in Section 7.4, "Performance Monitor Interrupt." 0 Performance monitor interrupts are disabled. 1 Performance monitor interrupts are enabled and occur when an enabled condition or event occurs.

Table 2-40. PMGC0 Field Descriptions (continued)

Bits	Name	Description
34	FCECE	Freeze counters on enabled condition or event Enabled conditions and events are described in Section 7.4, “Performance Monitor Interrupt.” 0 The PMCs can be incremented (if permitted by other PM control bits). 1 The PMCs can be incremented (if permitted by other PM control bits) only until an enabled condition or event occurs. When an enabled condition or event occurs, PMGC0[FAC] is set. It is up to software to clear FAC.
35–63	—	Reserved, should be cleared.

2.15.2 User Global Control Register 0 (UPMGC0)

The contents of PMGC0 are reflected to UPMGC0, which is read by user-level software. UPMGC0 is read with the **mfpmr** instruction using PMR384.

2.15.3 Local Control A Registers (PMLCa0–PMLCa3)

The local control A registers 0 through 3 (PMLCa0–PMLCa3) function as event selectors and give local control for the corresponding performance monitor counters. PMLCa works with the corresponding PMLCb register. PMLCa registers are shown in Figure 2-56.

	32	33	34	35	36	37	38	40	41		47	48		63
Field	FC	FCS	FCU	FCM1	FCM0	CE	—	EVENT						
Reset	All zeros													
R/W	PMLCa0–PMLCa3: Supervisor R/W UPMLCa0–UPMLCa3: User read-only													
PMR	PMLCa0: PMR144, PMLCa1: PMR145, PMLCa2: PMR146, PMLCa3: PMR147 UPMLCa0: PMR128, UPMLCa1: PMR129, UPMLCa2: PMR130, UPMLCa3: PMR131													

Figure 2-56. Local Control A Registers (PMLCa0–PMLCa3)
User Local Control A Registers (UPMLCa0–UPMLCa3)

Table 2-41 describes PMLCa fields.

Table 2-41. PMLCa0–PMLCa3 Field Descriptions

Bits	Name	Description
32	FC	Freeze counter 0 The PMC is incremented (if permitted by other PM control bits). 1 The PMC is not incremented.
33	FCS	Freeze counter in supervisor state 0 The PMC is incremented (if permitted by other PM control bits). 1 The PMC is not incremented if MSR[PR] = 0.
34	FCU	Freeze counter in user state 0 The PMC is incremented (if permitted by other PM control bits). 1 The PMC is not incremented if MSR[PR] = 1.

Table 2-41. PMLCa0–PMLCa3 Field Descriptions (continued)

Bits	Name	Description
35	FCM1	Freeze counter while mark = 1 0 The PMC is incremented (if permitted by other PM control bits). 1 The PMC is not incremented if MSR[PMM] = 1.
36	FCM0	Freeze counter while mark = 0 0 The PMC is incremented (if permitted by other PM control bits). 1 The PMC is not incremented if MSR[PMM] = 0.
37	CE	Condition enable 0 PMCx overflow conditions cannot occur (PMCx cannot cause interrupts, cannot freeze counters) 1 Overflow conditions occur when the most-significant-bit of PMCx is equal to 1. It is recommended that CE be cleared when counter PMCx is selected for chaining.
38–40	—	Reserved, should be cleared.
41–47	EVENT	Event selector. Up to 128 events selectable. See Section 7.7, “Event Selection.”
48–63	—	Reserved, should be cleared.

2.15.4 User Local Control A Registers (UPMLCa0–UPMLCa3)

The contents of PMLCa0–PMLCa3 are reflected to UPMLCa0–UPMLCa3, which are read by user-level software with **mfpmr** using PMR numbers in Table 2-39.

2.15.5 Local Control B Registers (PMLCb0–PMLCb3)

Local control B registers 0 through 3 (PMLCb0–PMLCb3) specify a threshold value and a multiple to apply to a threshold event selected for the corresponding performance monitor counter. For the e500, thresholding is supported only for PMC0 and PMC1. PMLCb works with the corresponding PMLCa. PMLCb registers are shown in Figure 2-57.

	32	52	53	55	56	57	58	63
Field	—			THRESHMUL	—		THRESHOLD	
Reset	All zeros							
R/W	PMLCb0–PMLCb3: Supervisor R/W UPMLCb0–UPMLCb3: User read-only							
PMR	PMLCb0: PMR272, PMLCb1: PMR273, PMLCb2: PMR274, PMLCb3: PMR275 UPMLCb0: PMR256, UPMLCb1: PMR257, UPMLCb2: PMR258, UPMLCb3: PMR259							

Figure 2-57. Local Control B Registers (PMLCb0–PMLCb3)
User Local Control B Registers (UPMLCb0–UPMLCb3)

Table 2-42 describes PMLCb fields.

Table 2-42. PMLCb0–PMLCb3 Field Descriptions

Bits	Name	Description
32–52	—	Reserved, should be cleared.
53–55	THRESHMUL	Threshold multiple 000 Threshold field is multiplied by 1 ($PMLCb_n[THRESHOLD] * 1$) 001 Threshold field is multiplied by 2 ($PMLCb_n[THRESHOLD] * 2$) 010 Threshold field is multiplied by 4 ($PMLCb_n[THRESHOLD] * 4$) 011 Threshold field is multiplied by 8 ($PMLCb_n[THRESHOLD] * 8$) 100 Threshold field is multiplied by 16 ($PMLCb_n[THRESHOLD] * 16$) 101 Threshold field is multiplied by 32 ($PMLCb_n[THRESHOLD] * 32$) 110 Threshold field is multiplied by 64 ($PMLCb_n[THRESHOLD] * 64$) 111 Threshold field is multiplied by 128 ($PMLCb_n[THRESHOLD] * 128$)
56–57	—	Reserved, should be cleared.
58–63	THRESHOLD	Threshold. Only events that exceed the THRESHOLD value are counted. Events to which a threshold value applies are implementation-dependent as are the dimension (for example duration in cycles) and the granularity with which the threshold value is interpreted. By varying the threshold value, software can obtain a profile of the event characteristics subject to thresholding. For example, if PMC1 is configured to count cache misses that last longer than the threshold value, software can obtain the distribution of cache miss durations for a given program by monitoring the program repeatedly using a different threshold value each time.

2.15.6 User Local Control B Registers (UPMLCb0–UPMLCb3)

The contents of PMLCb0–PMLCb3 are reflected to UPMLCb0–UPMLCb3, which are read by user-level software with **mfpmr** using PMR numbers in Table 2-39.

2.15.7 Performance Monitor Counter Registers (PMC0–PMC3)

The performance monitor counter registers PMC0–PMC3, shown in Figure 2-58, are 32-bit counters that can be programmed to generate interrupt signals when they overflow. Each counter is enabled to count 128 events.

	32	33	63
Field	OV	Counter value	
Reset	All zeros		
R/W	PMC0–PMC3: Supervisor R/W UPMC0–UPMC3: User read-only		
PMR	PMC0: PMR16, PMC1: PMR17, PMC2.: PMR18, PMC3: PMR19 UPMC0: PMR0, UPMC1: PMR1, UPMC2,: PMR2, UPMC3: PMR3		

Figure 2-58. Performance Monitor Counter Registers (PMC0–PMC3)
User Performance Monitor Counter Registers (UPMC0–UPMC3)

Table 2-43 describes PMC register fields.

Table 2-43. PMC0–PMC3 Field Descriptions

Bits	Name	Description
32	OV	Overflow. When this bit is set, it indicates this counter reaches its maximum value.
33–63	Counter Value	Indicates the number of occurrences of the specified event.

Counters overflow when the high-order bit (the sign bit) becomes set; that is, they reach the value 2,147,483,648 (0x8000_0000). However, an exception is not signaled unless PMGC0[PMIE] and PMLCax[CE] are also set as appropriate.

The interrupts are masked by clearing MSR[EE]. An interrupt that is signaled while MSR[EE] is zero is not taken until MSR[EE] is set. Setting PMGC0[FCECE] forces counters to stop counting when an enabled condition or event occurs.

Software is expected to use **mtpmr** to explicitly set PMCs to non-overflowed values. Setting an overflowed value may cause an erroneous exception. For example, if both PMGC0[PMIE] and PMLCax[CE] are set and the **mtpmr** loads an overflowed value into PMCx, an interrupt may be generated without an event counting having taken place.

PMC registers are accessed with **mtpmr** and **mfpmr** using PMR numbers in Table 2-38.

2.15.8 User Performance Monitor Counter Registers (UPMC0–UPMC3)

The contents of PMC0–PMC3 are reflected to UPMC0–UPMC3, which are read by user-level software with the **mfpmr** instruction using PMR numbers in Table 2-39.

2.16 Instruction Address Resume Register (IARR)

The IARR, shown in Figure 2-59, contains the address of the next instruction where fetching resumes after a softstop. It is read by user- and supervisor-level software and written using **mfspr** and **mtspr**.

	32	63
Field	Address of instruction after softstop	
Reset	All zeros	
R/W	R/W	
SPR	SPR 518	

Figure 2-59. Instruction Address Resume Register (IARR)

2.17 Nexus Process ID Register (NPIDR)

The NPIDR is shown in Figure 2-60. The NPIDR value is transferred to the NEXUS port whenever it is written

	32	63
Field	Nexus Process ID	
Reset	All zeros	
R/W	R/W	
SPR	SPR 517	

Figure 2-60. Nexus Process ID Register (NPIDR)

2.18 Book E SPR Model

This section describes SPR invalid references, synchronization requirements, and preserved, reserved, and allocated registers.

2.18.1 Invalid SPR References

System behavior when an invalid SPR is referenced depends on the privilege level of the register. The register privilege level is determined by bit 5 in the SPR address.

- If the invalid SPR is accessible in user mode (bit 5 of an SPR number = 0), an illegal instruction exception is taken.
- If the invalid SPR is accessible only in supervisor mode (bit 5 of an SPR number = 1) and the core complex is in supervisor mode ($MSR[PR] = 0$), the results of the attempted access are boundedly undefined.
- If the invalid SPR address is accessible only in supervisor mode (bit 5 of an SPR number = 1) and the core complex is not in supervisor mode ($MSR[PR] = 1$), a privilege exception is taken. These results are summarized in Table 2-44.

Table 2-44. System Response to an Invalid SPR Reference

SPR address bit 5	MSR[PR]	Response
0 (User)	x	Illegal exception
1 (Supervisor)	0 (Supervisor)	Boundedly undefined
	1 (User)	Privilege exception

2.18.2 Synchronization Requirements for SPRs

The software requirements for synchronization before and after accessing certain SPRs are shown in Table 2-45. Except for these registers, there are no synchronization requirements for accessing SPRs beyond those stated in Book E.

Table 2-45. Synchronization Requirements for SPRs

Registers	Instruction	Instruction Required Before	Instruction Required After
BBEAR	mtspr bbear	none	isync
BBTAR	mtspr bbtar	none	isync
BUCSR	mtspr BUCSR	none	isync
DBCR0	mtspr dbcr0	none	isync
DBCR1	mtspr dbcr1	none	isync
HID0	mtspr hid0	none	isync
HID1	mtspr hid1	none	isync
L1CSR0	mtspr l1csr0	msync, isync	isync
L1CSR1	mtspr l1csr1	none	isync
MAS[0-4,6]	mtspr mas[0-4,6]	none	isync
MMUCSR0	mtspr mmucsr0	none	isync
PID0-PID2	mtspr pid[0-2]	none	isync
SPEFSCR	mtspr SPEFSCR	none	isync

2.18.3 Preserved Special-Purpose Registers

Preserved SPR numbers are those that otherwise would be classified as reserved, but have legacy use which requires their deployment in the Book E to be deferred as long as possible in order to allow legacy hardware and software to migrate these legacy SPRs to Book E allocated SPR space.

Table 2-46. Preserved Special-Purpose Registers

Preserved SPR	Preserved SPR Numbers	
	Decimal	Binary
Book III DSISR	18	00000 1 0010
Book III DAR	19	00000 1 0011
Book III SDR1	25	00000 1 1001
8xx EIE	80	00010 1 0000
8xx EID	81	00010 1 0001
8xx NRE	82	00010 1 0010

Table 2-46. Preserved Special-Purpose Registers (continued)

Preserved SPR	Preserved SPR Numbers	
	Decimal	Binary
5xx,8xx CMPA	144	00100 1 0000
5xx,8xx CMPB	145	00100 1 0001
5xx,8xx CMPC	146	00100 1 0010
5xx,8xx CMPD	147	00100 1 0011
5xx,8xx ICR	148	00100 1 0100
5xx,8xx DER	149	00100 1 0101
5xx,8xx COUNTA	150	00100 1 0110
5xx,8xx COUNTB	151	00100 1 0111
5xx,8xx CMPE	152	00100 1 1000
5xx,8xx CMPF	153	00100 1 1001
5xx,8xx CMPG	154	00100 1 1010
5xx,8xx CMPH	155	00100 1 1011
5xx,8xx LCTRL1	156	00100 1 1100
5xx,8xx LCTRL2	157	00100 1 1101
5xx,8xx ICTRL	158	00100 1 1110
5xx,8xx BAR	159	00100 1 1111
Book III ASR	280	01000 1 1000
Book III EAR	282	01000 1 1010

2.18.4 Reserved SPRs

An SPR number in the range 0x000–0x1FF (0–511) that is not defined and is not preserved (see Table 2-46) is reserved.

2.18.5 Allocated SPRs

SPR numbers allocated for implementation-dependent use are 0x200–0x3FF (512–1023). Table 2-47 lists allocated SPRs implemented in the e500.

Table 2-47. Allocated SPRs Implemented in the e500

SPR	Mnemonic	Register
48	PID0 ¹	Process ID register 0. This is not truly an allocated SPR; however, Book E defines only this PID register and refers to it as PID rather than PID0.
512	SPEFSCR	Signal processing and embedded floating-point status and control register
513	BBEAR	Branch buffer entry address register
514	BBTAR	Branch buffer target address register
515	L1CFG0	L1 cache configuration register 0
516	L1CFG1	L1 cache configuration register 1
517	NPIDR ²	Nexus processor ID register
518	IARR ³	Instruction address resume register
528	IVOR32	SPE APU unavailable exception

Table 2-47. Allocated SPRs Implemented in the e500 (continued)

SPR	Mnemonic	Register
529	IVOR33	SPE floating-point data exception
530	IVOR34	SPE floating-point round exception
531	IVOR35	Performance monitor Interrupt vector offset register
570	MCSRR0	Machine-check save/restore register 0
571	MCSRR1	Machine-check save/restore register 1
572	MCSR	Machine check syndrome register
573	MCAR	Machine check address register
624	MAS0	MMU assist register 0
625	MAS1	MMU assist register 1
626	MAS2	MMU assist register 2
627	MAS3	MMU assist register 3
628	MAS4	MMU assist register 4
629	MAS5	MMU assist register 5. Defined by Motorola Book E but not implemented in the e500
630	MAS6	MMU assist register 6
633	PID1 ³	Process ID register 1
634	PID2 ³	Process ID register 2
688	TLB0CFG	TLB configuration register 0
689	TLB1CFG	TLB configuration register 1
1008	HID0	Hardware implementation dependent reg 0
1009	HID1	Hardware implementation dependent reg 1
1010	L1CSR0	L1 cache control and status register 0
1011	L1CSR1	L1 cache control and status register 1
1012	MMUCSR0	MMU control and status register 0
1013	BUCSR	Branch unit control and status register
1015	MMUCFG	MMU configuration register

¹ An update to PID0, PID1, or PID2 must always be followed by an **isync**.

² The NPIDR register is transferred to the NEXUS port whenever it is written.

³ IARR contains the address of the next instruction to resume after a softstop.

Chapter 3

Instruction Model

The e500 core complex is a 32-bit implementation of the Book E architecture as defined in the Book E architecture specification. This architecture specification allows for different processor implementations, which implementations may provide clarifications, extensions, or deviations from the architectural descriptions. This chapter provides information about the Book E architecture as it relates specifically to the e500 core complex. The e500 core complex also implements several auxiliary processing units (APUs), which define additional instructions, registers, and interrupts. The APUs are described in detail in Chapter 10, “Auxiliary Processing Units (APUs).”

3.1 Operand Conventions

This section describes operand conventions as they are represented in the Book E architecture. These conventions follow the basic descriptions in the classic PowerPC architecture with some changes in terminology. For example, distinctions between user and supervisor-level instructions are maintained, but the designations—UISA, VEA, and OEA—do not apply. Detailed descriptions are provided of conventions used for storing values in registers and memory, accessing processor registers, and representing data in these registers.

3.1.1 Data Organization in Memory and Data Transfers

Bytes in memory are numbered consecutively starting with 0. Each number is the address of the corresponding byte.

Memory operands can be bytes, half words, words, or double words or, for the load/store multiple instruction type, a sequence of bytes or words. The address of a memory operand is the address of its first byte (that is, of its lowest-numbered byte). Operand length is implicit for each instruction.

3.1.2 Alignment and Misaligned Accesses

The operand of a single-register memory access instruction has an alignment boundary equal to its length. An operand’s address is misaligned if it is not a multiple of its width.

Unsupported Instructions and Instruction Forms

The concept of alignment is also applied more generally to data in memory. For example, a 12-byte data item is said to be word-aligned if its address is a multiple of four.

Some instructions require their memory operands to have certain alignment. In addition, alignment can affect performance. For single-register memory access instructions, the best performance is obtained when memory operands are aligned.

Instructions are 32 bits (one word) long and must be word-aligned.

Memory operands for single-register memory access instructions have the characteristics described in Table 3-1.

Table 3-1. Address Characteristics of Aligned Operands

Operand	Operand Length	Addr[60–63] if aligned
Byte	8 bits	xxxx ¹
Half word	2 bytes	xxx0
Word	4 bytes	xx00
Double word	8 bytes	x000

¹ An x in an address bit position indicates that the bit can be 0 or 1 independent of the state of other bits in the address.

The concept of alignment is also applied more generally to any data element in memory. For example, a 12-byte data element is said to be word-aligned if its address is an integral multiple of 4.

3.1.3 e500 Floating-Point Implementation

The e500 does not implement the floating-point instructions as they are defined in Book E. Attempts to execute a Book E-defined floating-point instruction results in an illegal instruction exception. However, the SPE APU supports single-precision vector floating-point operations and a separate instruction subset that performs single-precision floating-point operations using the lower 32 bits of the GPRs. These instructions are described in Section 3.11.1.2, “SPE APU Floating-Point Instructions.” Unlike the PowerPC UISA, the SPE APU stores floating-point values as single-precision values in true 32-bit, single-precision format rather than in a 64-bit double-precision format used with FPRs. See Section Chapter 10, “Auxiliary Processing Units (APUs),” for more details.

3.2 Unsupported Instructions and Instruction Forms

Because the core complex uses a 32-bit Book E core, all of the instructions defined for 64-bit implementations of the Book E architecture are illegal on the e500. These instructions are listed in Table 3-3. The core complex takes an illegal instruction exception type program interrupt upon encountering a 64-bit Book E instruction.

NOTE

Extended addressing forms of all load and store instructions are illegal because they calculate a 64-bit effective address. Also, except for certain SPE APU vector instructions, all double-word instruction forms are illegal because only 64-bit implementations allow double-word operands.

The core complex does not support the Book E instructions listed in Table 3-2 and Table 3-3. An illegal instruction exception is generated if the processor attempts to execute one of these instructions. Some instructions have the following optional features indicated by square brackets:

- Condition register (CR) update—The dot (.) suffix on the mnemonic enables the update of the CR.
- Overflow option—The **o** suffix indicates that the overflow bit in the XER is enabled.

Table 3-2 lists 32-bit instructions that are not implemented on the e500.

Table 3-2. Unsupported Book E Instructions (32-Bit)

Name	Mnemonic
Floating Absolute Value [and record CR]	fabs [.]
Floating Add [Single] [and record CR]	fadd [s][.]
Floating Convert From Integer Double Word	fcfd
Floating Compare Ordered	fcmpo
Floating Compare Unordered	fcmpu
Floating Convert To Integer Double Word	fctid
Floating Convert To Integer Double Word [and round to Zero]	fctid [z]
Floating Convert To Integer Word [and round to Zero] [and record CR]	fctiw [z][.]
Floating Divide [Single] [and record CR]	fdiv [s][.]
Floating Multiply-Add [Single] [and record CR]	fmadd [s][.]
Floating Move Register [and record CR]	fmr [.]
Floating Multiply-Subtract [Single] [and record CR]	fmsub [s][.]
Floating Multiply [single] [and record CR]	fmul [s][.]
Floating Negative Absolute Value [and record CR]	fnabs [.]
Floating Negate [and record CR]	fneg [.]
Floating Negative Multiply-Add [Single] [and record CR]	fnmadd [s][.]
Floating Negative Multiply-Subtract [Single] [and record CR]	fnmsub [s][.]
Floating Reciprocal Estimate Single [and record CR]	fres [.]
Floating Round to Single-Precision [and record CR]	frsp [.]
Floating Reciprocal Square Root Estimate [and record CR]	frsqre [.]

Table 3-2. Unsupported Book E Instructions (32-Bit) (continued)

Name	Mnemonic
Floating Select [and record CR]	fsel [.]
Floating Square Root [Single] [and record CR]	fsqrt [s][.]
Floating Subtract [Single] [and record CR]	fsub [s][.]
Load Floating-Point Double [with Update] [Indexed] [Extended]	lfd [u][x][e]
Load Floating-Point Single [with Update] [Indexed] [Extended]	lfs [u][x][e]
Load String Word Immediate	lswi
Load String Word Indexed	lswx
Move From Device Control Register	mfdcr
Move From FPSCR [and record CR]	mffs [.]
Move To Device Control Register	mtdcr
Move To FPSCR Bit 0 [and record CR]	mtfsb0 [.]
Move To FPSCR Bit 1 [and record CR]	mtfsb1 [.]
Move To FPSCR Field [Immediate] [and record CR]	mtfsf [i][.]
Store Floating-Point Double [with Update] [Indexed] [Extended]	stfd [u][x][e]
Store Floating-Point as Integer Word Indexed [Extended]	stfiwx [e]
Store Floating-Point Single [with Update] [Indexed] [Extended]	stfs [u][x][e]
Store String Word Immediate	stswi
Store String Word Indexed	stswx

Table 3-3 lists Book E instructions that are not supported because they are 64-bit instructions.

Table 3-3. Unsupported Book E Instructions (64-Bit)

Name	Mnemonic
Add Extended with CA64 [and record OV]	adde64 [o]
Add to Minus One Extended with CA64 [and record OV]	addme64 [o]
Add to Zero Extended with CA64 [and record OV]	addze64 [o]
Branch Conditional to Count Register Extended [and Link]	bcctre [l]
Branch Conditional Extended [and Link] [Absolute]	bce [l][a]
Branch Conditional to Link Register Extended [and Link]	bcldre [l]
Branch Extended [and Link] [Absolute]	be [l][a]
Compare [Logical] [Immediate] (L=1 indicates 64-bit operands)	cmp [l][i] (L = 1)
Count Leading Zeros Double Word	cntlzd
Data Cache Block Allocate Indexed Extended	dcbae

Table 3-3. Unsupported Book E Instructions (64-Bit) (continued)

Name	Mnemonic
Data Cache Block Flush Indexed Extended	dcbfe
Data Cache Block Invalidate Extended	dcbie
Data Cache Block Store Indexed Extended	dcbste
Data Cache Block Touch Indexed Extended	dcbte
Data Cache Block Touch and Lock Set Extended	dcbtlse
Data Cache Block Touch for Store Indexed Extended	dcbtste
Data Cache Block set to Zero Indexed Extended	dcbze
Divide Double Word [Unsigned]	divd[o][.]
Instruction Cache Block Invalidate Indexed Extended	icbie
Instruction Cache Block Touch Indexed Extended	icbte
Load Byte and Zero [with Update] [Indexed] Extended	lbz[u][x]e
Load Double Word and Reserve Indexed Extended	ldarxe
Load Double Word [with Update] [Indexed] [Extended]	ld[u][x][e]
Load Half Word Algebraic [with Update] [Indexed] Extended	lha[u][x]e
Load Half Word Byte-Reverse Indexed Extended	lhbrxe
Load Half Word and Zero [with Update] [Indexed] Extended	lhz[u][x]e
Load Word and Reserve Indexed Extended	lwarxe
Load Word Byte-Reverse Indexed Extended	lwbrxe
Load Word and Zero [with Update] [Indexed] Extended	lwz[u][x]e
Multiply High Double Word [Unsigned]	mulhd[u]
Multiply Low Double Word [and record OV]	muld[o]
Rotate Left Double Word Immediate then Clear [and record CR]	rldic[.]
Rotate Left Double Word Immediate then Mask Insert [and record CR]	rdimif[.]
Rotate Left Double Word [Immediate] then Clear Left [and record CR]	rl[d i]cl[.]
Rotate Left Double Word [Immediate] then Clear Right [and record CR]	rl[d i]cr[.]
Shift Left Double Word [and record CR]	sld[.]
Shift Right Algebraic Double Word [Immediate] [and record CR]	srad[i][.]
Shift Right Double Word [and record CR]	srd[.]
Store Byte [with Update] [Indexed] Extended	stb[u][x]e
Store Double Word Conditional Indexed [Extended]	stdcxe.
Store Double Word [with Update] [Indexed] [Extended]	std[u][x][e]
Store Half Word Byte-Reverse Indexed Extended	sthbrxe
Store Half Word [with Update] [Indexed] Extended	sth[u][x]e

Table 3-3. Unsupported Book E Instructions (64-Bit) (continued)

Name	Mnemonic
Store Word Byte-Reverse Indexed Extended	stwbrxe
Store Word Conditional Indexed Extended [and record CR]	stwcxe[.]
Store Word [with Update] [Indexed] Extended	stw[u][x]e
Subtract From Extended with CA64 [and record OV]	subfe64[o]
Subtract From Minus One Extended with CA64 [and record OV]	subfme64[o]
Subtract From Zero Extended with CA64 [and record OV]	subfze64[o]
Trap Double Word [Immediate]	td[i]
TLB Invalidate Virtual Address Indexed Extended	tlbivaxe
TLB Search Indexed Extended	tlbsxe

3.3 Instruction Set Summary

This chapter describes instructions and addressing modes defined for the e500. These instructions are divided into the following functional categories:

- Integer instructions—These include arithmetic and logical instructions. For more information, see Section 3.4.1.1, “Integer Instructions.”
- Floating-point instructions—These include floating-point vector and scalar arithmetic instructions implemented in the SPE APU. See Section 3.11.1.2, “SPE APU Floating-Point Instructions.” The e500 does not support Book E–defined floating-point instructions or floating-point registers.
- Load and store instructions—For more information, see Section 3.4.1.2, “Load and Store Instructions.”
- Flow control instructions—These include branching instructions, condition register logical instructions, trap instructions, and other instructions that affect the instruction flow. For more information, see Section 3.4.1.3, “Branch and Flow Control Instructions.”
- Processor control instructions—These instructions are used for synchronizing memory accesses. For more information, see Section 3.4.1.6, “Processor Control Instructions.”
- Memory synchronization instructions—These instructions are used for memory synchronizing. See Section 3.4.1.7, “Memory Synchronization Instructions,” for more information.
- Memory control instructions—These instructions provide control of caches and TLBs. For more information, see Section 3.4.1.9, “Memory Control Instructions,” and Section 3.4.2.2, “Supervisor-Level Memory Control Instructions.”

- Signal processing instructions—These include a set of vector arithmetic and logic instructions optimized for signal processing tasks. See Section 3.11.1, “Signal Processing Engine (SPE) APU Instructions.”

Note that instruction groupings used here do not indicate the execution unit that processes a particular instruction or group of instructions. This information, which is useful for scheduling instructions most effectively, is provided in Chapter 4, “Execution Timing.”

Integer instructions operate on word operands. The PowerPC architecture uses instructions that are 4 bytes long and word-aligned. It provides for byte, half-word, and word operand loads and stores between memory and a set of 32 general-purpose registers (GPRs).

Arithmetic and logical instructions do not read or modify memory. To use the contents of a memory location in a computation and then modify the same or another location, the memory contents must be loaded into a register, modified, and then written to the target location using load and store instructions.

The description of each instruction includes the mnemonic and a formatted list of operands. To simplify assembly language programming, a set of simplified mnemonics and symbols is provided for some of the frequently used instructions; see Appendix A, “Simplified Mnemonics,” for a complete list of simplified mnemonics. Programs written to be portable across the various assemblers for the PowerPC architecture should not assume the existence of mnemonics not described in that document.

3.3.1 Classes of Instructions

The e500 instructions belong to one of the following four classes:

- Defined instructions (see Section 3.3.1.2, “Defined Instruction Class.”)
- Allocated instructions (see Section 3.3.1.3, “Allocated Instruction Class.”)
- Preserved instructions (see Section 3.3.1.4, “Preserved Instruction Class.”)
- Reserved (illegal or no-op) instructions (see Section 3.3.1.5, “Reserved Instruction Class.”)

The class is determined by examining the primary opcode and the extended opcode, if any. If the opcode, or combination of opcode and extended opcode, is not that of a defined instruction, preserved, or reserved instruction, the instruction is illegal.

Instruction encodings that are now illegal can become assigned to instructions in the architecture or can be reserved by being assigned to processor-specific instructions.

3.3.1.1 Definition of Boundedly Undefined

If instructions are encoded with incorrectly set bits in reserved fields, the results on execution can be said to be boundedly undefined. If a user-level program executes the incorrectly coded instruction, the resulting undefined results are bounded in that a spurious

change from user to supervisor state is not allowed, and the level of privilege exercised by the program in relation to memory access and other system resources cannot be exceeded. Boundedly undefined results for a given instruction can vary between implementations and between execution attempts in the same implementation.

3.3.1.2 Defined Instruction Class

This class of instructions consists of all the instructions defined in Book E. In general, defined instructions are guaranteed to be supported within a Book E system as specified by the architecture, either within the processor implementation itself or within emulation software supported by the system operating software.

For implementations that only provide the 32-bit subset of Book E such as the e500 core, emulation of the 64-bit behavior of the defined instructions is not supported. See Appendix C, “Guidelines for 32-Bit Book E.”

Any attempt to execute a defined instruction results in one of the following events:

- An illegal instruction exception type program interrupt, if the instruction is not recognized by the implementation.
- An unimplemented instruction exception type program interrupt, if the instruction is recognized but not supported by the implementation, and is not a floating-point instruction.
- An unimplemented instruction exception type program interrupt, if the instruction is recognized but not supported by the implementation, and is a floating-point instruction and floating-point processing is enabled.
- The floating-point unavailable interrupt (not implemented on the e500) if the instruction is recognized but is not supported by the implementation or is a floating-point instruction and floating-point processing is disabled.
- The floating-point unavailable interrupt (not implemented on the e500) when floating-point processing is disabled and a floating-point instruction is recognized and is not supported by the implementation.
- The processor performs the actions described in the rest of this document, if the instruction is recognized and supported by the implementation. The architected behavior may cause other exceptions.

A defined instruction may be retained by future versions of Book E as a defined instruction, or may be reclassified as a preserved instruction (process of removal from the architecture) and eventually classified as reserved-illegal.

3.3.1.3 Allocated Instruction Class

This class of instructions contains the set of instructions (a set of primary opcodes, as well as a set of extended opcodes for certain primary opcodes) used for implementation-specific instructions. Table 3-4 lists blocks of opcodes allocated for implementation-dependent use.

Table 3-4. Allocated Instructions

Primary Opcode	Extended Opcodes
0	All instruction encodings (bits 6–31) except 0x0000_0000 ¹ .
4	All instruction encodings (bits 6–31) SPE instructions
19	Extended opcodes (bits 21–30) --- 0buuuuu_0u11u
31	Extended opcodes (bits 21–30) --- uuuuu_0u11u The e500 uses this allocated space for the following instructions: dcblc 01100_00110 dcbtls 00101_00110 dcbtstls 00100_00110 icblc 00111_00110 icbtls 01111_00110 bblcls 10000_00110 bblcr 10001_00110 isel crb_01111 mfpmr 01010_01110 mtpmr 01110_01110
59	Extended opcodes (bits 21–30) --- uuuuu_0u10u
63	Extended opcodes (bits 21–30) --- uuuuu_0u10u (except 00000_01100 frsp)

¹ Instruction encoding 0x0000_0000 is and always will be reserved-illegal.

Allocated instructions are allocated to purposes that are outside the scope of Book E for implementation-dependent and application-specific use.

Any attempt to execute an allocated instruction results in one of the following events:

- If the instruction is not recognized by the implementation, an illegal instruction exception type program interrupt.
- An unimplemented instruction exception type program interrupt, if the instruction is recognized and enabled for execution but direct execution of the instruction is not supported by the implementation. This option may be used to allow emulation for unsupported allocated instructions.
- If the allocated instruction is implemented, the processor performs the actions described in the user's manual. The implementation-dependent behavior may cause other exceptions, which may cause other interrupts).

An allocated instruction is guaranteed by Book E to remain allocated.

NOTE

Some allocated instructions may have associated new process state, and, therefore, may provide an associated enable bit, similar in function to MSR[FP] for floating-point instructions. To be enabled for execution for these instructions implies that any associated enable bit is set to allow, or enable, execution of these instructions. For these allocated instructions, the architecture provides an auxiliary processor unavailable interrupt vector (not supported on the e500) in the event execution of any of these instructions is attempted when not enabled for execution.

Other allocated instructions may not have any associated new state and therefore may not require an associated enable bit. These instructions are assumed to always be enabled for execution if they are supported by the implementation.

3.3.1.4 Preserved Instruction Class

The preserved instruction class supports backward compatibility with the PowerPC Architecture. An attempt to execute a preserved instruction results in one of the following:

- If the implementation does not recognize the instruction, an illegal instruction exception type program interrupt occurs.
- If the instruction is recognized and supported by the implementation, the processor performs the actions defined in the previous version of the architecture.

Future versions of Book E may retain a preserved instruction as a preserved instruction, may reclassify it as an allocated instruction, or may adopted it as part of Book E.

Preserved opcodes are listed in Table 3-5.

Table 3-5. Preserved Instructions

Primary Opcode	Extended Opcodes
0	No preserved extended opcodes
4	No preserved extended opcodes
19	No preserved extended opcodes

Table 3-5. Preserved Instructions (continued)

Primary Opcode	Extended Opcodes
31	Extended opcodes (bits 21–30) 210 0b00110_10010 (mtsr) 242 0b00111_10010 (mtsrin) 370 0b01011_10010 (tlbia) 306 0b01001_10010 (tlbie) 371 0b01011_10011 (mftb) 595 0b10010_10011 (mfsr) 659 0b10100_10011 (mfsrin) 310 0b01001_10110 (eciwx) 438 0b01101_10110 (ecowx)
59	No preserved extended opcodes
63	No preserved extended opcodes

3.3.1.5 Reserved Instruction Class

This class of instructions consists of all instruction primary opcodes (and associated extended opcodes, if applicable) that do not belong to either the defined, allocated, or preserved instruction classes.

Reserved instructions are available for future extensions of Book E. That is, some future version of Book E may define any of these instructions to perform new functions or make them available for implementation-dependent use as allocated instructions. There are two types of reserved instructions, reserved-illegal and reserved-nop.

Attempts to execute a reserved-illegal instruction cause an illegal instruction exception type program interrupt (see Section 5.7.7, “Program Interrupt”) on implementations conforming to the current version of Book E. Reserved-illegal instructions are, therefore, available for future extensions to Book E which would affect architected state. Such extensions might include new forms of integer or floating-point arithmetic or new forms of load or store instructions which write their result in an architected register.

Attempts to execute a reserved-nop instruction either has no effect on implementations conforming to the current version of Book E (that is, treated as a no-operation instruction), or causes an illegal instruction exception type program interrupt (see Section 5.7.7, “Program Interrupt”). Reserved-nop instructions are available for future architecture extensions which have no effect on architected state. Such extensions might include performance-enhancing hints such as new forms of cache touch instructions, and could be added while remaining functionally compatible with implementations of previous versions of Book E.

A reserved-illegal instruction may be retained by future versions of Book E as a reserved-illegal instruction, may be subsequently reclassified as an allocated instruction, or may even be employed in the role of a subsequently defined instruction.

A reserved-nop instruction may be retained by future versions of Book E as a reserved-nop instruction, may be subsequently reclassified as an allocated instruction, or may even be employed in the role of a subsequently defined instruction which has no effect on architected state.

3.3.2 Instruction Forms

This section describes preferred instruction forms, addressing modes, and synchronization.

3.3.2.1 Preferred Instruction Forms

The Or Immediate (**ori**) instruction has a preferred form used for expressing a no-op.

The preferred no-op is **ori 0,0,0**

3.3.2.2 Invalid Instruction Forms

Some of the defined instructions have invalid forms. An instruction form is invalid if one or more fields of the instruction, excluding the opcode field(s), are coded incorrectly in a manner that can be deduced by examining only the instruction encoding.

Any attempt to execute an invalid form of an instruction either causes an illegal instruction type program interrupt or yield boundedly undefined results. Exceptions to this rule are stated in the instruction descriptions.

Some kinds of invalid form instructions can be deduced just from examining the instruction layout. These are listed below.

- Field shown as reserved but coded as nonzero.
- Field shown as containing a particular value but coded as some other value.

These invalid forms are not discussed further.

Other invalid instruction forms can be deduced by detecting an invalid encoding of one or more of the instruction operand fields. These kinds of invalid form are identified in the instruction descriptions.

- Branch conditional and branch conditional extended instructions (undefined encoding of BO field)
- Load with update instructions ($rD = rA$ or $rA = 0$)
- Store with update instructions ($rA = 0$)
- Load multiple instruction (rA or rB in range of registers to be loaded)

3.3.3 Addressing Modes

This section provides an overview of conventions for addressing memory and for calculating effective addresses as defined by the Book E architecture for 32-bit implementations.

3.3.3.1 Memory Addressing

A program references memory using the effective address computed by the processor when it executes a memory access or branch instruction (or certain other instructions described in Section 3.4.1.9.1, “User-Level Cache Instructions,” and Section 3.4.2.2.1, “Supervisor-Level Cache Instruction,” or when it fetches the next sequential instruction.

3.3.3.2 Memory Operands

Bytes in memory are numbered consecutively starting with 0. Each number is the address of the corresponding byte.

Memory operands may be bytes, half words, words or, for the load/store multiple sequence of words or bytes. The address of a memory operand is the address of its first byte (that is, of its lowest-numbered byte). Byte ordering can be either big-endian or little-endian (see Section 3.3.3.4, “Byte Ordering”). The default byte and bit ordering is big-endian.

Operand length is implicit for each instruction with respect to memory alignment. The operand of a scalar memory access instruction has a natural alignment boundary equal to the operand length. In other words, the natural address of an operand is an integral multiple of the operand length. A memory operand is said to be aligned if it is aligned at its natural boundary; otherwise it is said to be misaligned. For more information about alignment, see Section 3.1.2, “Alignment and Misaligned Accesses.”

3.3.3.3 Effective Address Calculation

The 32-bit address computed by the processor when executing a memory access or branch instruction (or certain other instructions described in Section 3.4.1.9.1, “User-Level Cache Instructions,” Section 3.4.2.2.1, “Supervisor-Level Cache Instruction,” and Section 3.4.2.2.2, “Supervisor-Level TLB Management Instructions”), or when fetching the next sequential instruction, is called the effective address and specifies a byte in memory. For a memory access instruction, if the sum of the effective address and the operand length exceeds the maximum effective address, the memory access is considered to be undefined.

Effective address arithmetic, except for next sequential instruction address computations, wraps around from the maximum address, $2^{32}-1$, to address 0.

3.3.3.3.1 Data Memory Addressing Modes

Book E supports the following data memory addressing modes.

- Base+displacement addressing mode—The 16-bit D field is sign-extended and added to the contents of the GPR designated by **rA** or to zero if **rA** = 0. Instructions that use this addressing mode are of the D instruction format.”
- Base+index addressing mode—The contents of the GPR designated by **rB** are added to the contents of the GPR designated by **rA** or to zero if **rA** = 0. Instructions that use this addressing mode are of the X instruction format.”

In addition, APUs may provide additional addressing modes.

3.3.3.3.2 Instruction Memory Addressing Modes

Instruction memory addressing modes are as follows:

- I-form branch instructions—The 24-bit LI field is concatenated on the right with 0b00, sign-extended, and then added to either the address of the branch instruction if AA = 0, or to 0 if AA = 1.
- Taken B-form branch instructions—The 14-bit BD field is concatenated on the right with 0b00, sign-extended, and then added to either the address of the branch instruction if AA = 0, or to 0 if AA = 1.
- Taken XL-form branch instructions—The contents of bits LR[32–61] or CR[32–61] are concatenated on the right with 0b00.
- Sequential instruction fetching (or non-taken branch instructions)—The value 4 is added to the address of the current instruction to form the 32-bit effective address of the next instruction. If the address of the current instruction is 0xFFFF_FFFC, the address of the next sequential instruction is undefined.
- Any branch instruction with LK = 1—The value 4 is added to the address of the current instruction and the 32-bit result is placed into the LR. If the address of the current instruction is 0xFFFF_FFFC, the result placed into the LR is undefined.

Although some implementations may support next sequential instruction address computations wrapping from the highest address 0xFFFF_FFFC to 0x0000_0000 as part of the instruction flow, users are strongly encouraged not to depend on this behavior. Doing so can reduce the portability of their software. If code must span this boundary, software should place a non-linking branch at address 0xFFFF_FFFC, which always branches to address 0x0000_0000 (either absolute or relative branches work). See also Section Appendix C, “Guidelines for 32-Bit Book E.”

3.3.3.4 Byte Ordering

If scalars (individual data items and instructions) were indivisible, there would be no such concept as byte ordering. It is meaningless to consider the order of bits or groups of bits within the smallest addressable unit of memory, because nothing can be observed about such order. Only when scalars, which the programmer and processor regard as indivisible quantities, can comprise more than one addressable unit of memory does the question of order arise.

For a machine in which the smallest addressable unit of memory is the 64-bit double word, there is no question of the ordering of bytes within double words. All transfers of individual scalars between registers and memory are of double words, and the address of the byte containing the high-order 8 bits of a scalar is no different from the address of a byte containing any other part of the scalar.

For Book E, as for most computer architectures currently implemented, the smallest addressable unit of memory is the 8-bit byte. Many scalars are half words and words (double words in 64-bit implementations) which consist of groups of bytes. When a word-length scalar is moved from a register to memory, the scalar occupies four consecutive byte addresses. It thus becomes meaningful to discuss the order of the byte addresses with respect to the value of the scalar: which byte contains the highest-order eight bits of the scalar, which byte contains the next-highest-order 8 bits, and so on.

Given a scalar that contains multiple bytes, the choice of byte ordering is essentially arbitrary. There are $4! = 24$ ways to specify the ordering of 4 bytes within a word but only two of these orderings are sensible:

- The ordering that assigns the lowest address to the highest-order (left-most) 8 bits of the scalar, the next sequential address to the next-highest-order eight bits, and so on. This ordering is called big-endian because the big end (most-significant) of the scalar, considered as a binary number, comes first in memory. The Motorola 68000 is an example of a processor using this byte ordering.
- The ordering that assigns the lowest address to the lowest-order (right-most) 8 bits of the scalar, the next sequential address to the next-lowest-order eight bits, and so on. This ordering is called little-endian because the little end (least-significant) of the scalar, considered as a binary number, comes first in memory. The Intel 8086 is an example of a processor using this byte ordering.

Book E provides support for both big- and little-endian byte ordering in the form of a memory attribute. See Section 11.2.4, “Permission Attributes,” and Section 11.1.1.5, “Endianness.”

3.3.3.5 Synchronization Requirements

This section discusses synchronization requirements for special registers and TLBs. The synchronization described in this section refers to the state of the processor that is performing the synchronization.

Changing the value in certain system registers and invalidating TLB entries can have the side effect of altering the context in which data addresses and instruction addresses are interpreted, and in which instructions are executed. For example, changing MSR[IS]=0 to MSR[IS]=1 has the side effect of changing address space. These side effects need not occur in program order (program order refers to the execution of instructions in the strict order in which they occur in the program) and therefore may require explicit synchronization by software.

An instruction that alters the context in which data addresses or instruction addresses are interpreted, or in which instructions are executed, is called a context-altering instruction. This section covers all of the context-altering instructions. The software synchronization required for each is shown in Table 3-6 and Table 3-7.

A context-synchronizing interrupt (that is, any interrupt except non-recoverable machine check) can be used instead of a context-synchronizing instruction. If it is, references in this section to the synchronizing instruction should be interpreted as meaning the instruction at which the interrupt occurs. If no software synchronization is required either before or after a context-altering instruction, the phrase, ‘the synchronizing instruction before (or after) the context-altering instruction’ should be interpreted as meaning the context-altering instruction itself.

The synchronizing instruction before the context-altering instruction ensures that all instructions up to and including that synchronizing instruction are fetched and executed in the context that existed before the alteration. The synchronizing instruction after the context-altering instruction ensures that all instructions after that synchronizing instruction are fetched and executed in the context established by the alteration. Instructions after the first synchronizing instruction, up to and including the second synchronizing instruction, may be fetched or executed in either context.

If a sequence of instructions contains context-altering instructions and contains no instructions that are affected by any of the context alterations, no software synchronization is required within the sequence.

Sometimes advantage can be taken of the fact that certain instructions that occur naturally in the program, such as the **rfi** at the end of an interrupt handler, provide the required synchronization.

No software synchronization is required before altering the MSR (except perhaps when altering the WE bit: see the tables), because **mtmsr** is execution synchronizing. No software synchronization is required before most other alterations shown in Table 3-7, because all instructions before the context-altering instruction are fetched and decoded

before the context-altering instruction is executed (the processor must determine whether any of the preceding instructions are context-synchronizing)

Table 3-6 identifies the software synchronization requirements for data access for all context-altering instructions.

Table 3-6. Data Access

Context Altering Instruction or Event	Required Before	Required After	Notes
Interrupt	None	None	
rfi	None	None	
rfci	None	None	
sc	None	None	
mtmsr (PR)	None	CSI ¹	
mtmsr (ME)	None	CSI ¹	2
mtmsr (DS)	None	CSI ¹	
mtspr (DAC1, DAC2, DVC1, DVC2)	—	—	3
mtspr (DBCR0, DBCR2)	—	—	3
mtspr (DBSR)	—	—	3
mtspr (PID)	CSI ¹	CSI ¹	
tlbivax	CSI ¹	CSI ¹ and possibly msync	4, 5
tlbwe	CSI ¹	CSI ¹ and possibly msync	4,5

¹ CSI indicates any context-synchronizing instruction (that is, **sc**, **isync**, **rfci**, or **rfi**).

² A context-synchronizing instruction is required after altering MSR[ME] to ensure that the alteration takes effect for subsequent machine check interrupts, which may not be recoverable and therefore may not be context-synchronizing.

³ Synchronization requirements for changing any of the debug facility registers are implementation-dependent.

⁴ For data accesses, the context-synchronizing instruction before **tlbwe** or **tlbivax** ensures that all memory accesses due to preceding instructions have completed to a point at which they have reported all exceptions they will cause.

⁵ The context-synchronizing instruction after **tlbwe** or **tlbivax** ensures that subsequent accesses (data and instruction) use the updated value in any TLB entries affected. It does not ensure that all accesses previously translated by TLB entries being updated have completed with respect to memory; if these completions must be ensured, **tlbwe** or **tlbivax** must be followed by an **msync** and by a context-synchronizing instruction.

Table 3-7 below identifies the software synchronization requirements for instruction fetch and/or execution for all context-altering instructions.

Table 3-7. Instruction Fetch and/or Execution

Context Altering Instruction or Event	Required Before	Required After	Notes
Interrupt	None	None	
rfi	None	None	
rfci	None	None	
sc	None	None	

Table 3-7. Instruction Fetch and/or Execution (continued)

Context Altering Instruction or Event	Required Before	Required After	Notes
mtmsr (WE)	—	—	1
mtmsr (CE)	None	None	2
mtmsr (EE)	None	None	2
wrttee	None	None	2
wrtteei	None	None	2
mtmsr (PR)	None	CSI ³	
mtmsr (FP)	None	CSI ³	
mtmsr (ME)	None	CSI ³	1 ⁴
mtmsr (FE0)	None	CSI ³	
mtmsr (FE1)	None	CSI ³	
mtmsr (DE)	None	CSI ³	
mtspr (IVPR)	None	None	
mtspr (IVORi)	None	None	
mtmsr (IS)	None	CSI ³	5
mtspr (PID)	None	CSI ³	5
mtspr (DEC)	None	None	6
mtspr (TCR)	None	None	6
mtspr (TSR)	None	None	6
mtspr (IAC1, IAC2, IAC3, IAC4)	—	—	7
mtspr (DBCR0, DBCR1)	—	—	7
mtspr (DBSR)	—	—	7
tlbivax	None	CSI ³ or msync	8, 9
tlbwbe	None	CSI ³ or msync	8,9

¹ Synchronization requirements for changing the wait state enable are implementation-dependent.

² The effect of changing MSR[EE] or MSR[CE] is immediate.

If **mtmsr**, **wrttee**, or **wrtteei** clears MSR[EE], an external input, DEC or fixed-interval timer interrupt does not occur after the instruction is executed.

If **mtmsr**, **wrttee**, or **wrtteei** changes MSR[EE] from 0 to 1 when an external input, decremter, fixed-interval timer, or higher priority enabled exception exists, the corresponding interrupt occurs immediately after the **mtmsr**, **wrttee**, or **wrtteei** is executed, and before the next instruction is executed in the program that set MSR[EE].

³ CSI indicates any context-synchronizing instruction (that is, **sc**, **isync**, **rftci**, **rftmci**, or **rfti**).

⁴ A context-synchronizing instruction is required after altering MSR[ME] to ensure that the alteration takes effect for subsequent machine check interrupts, which may not be recoverable and so may not be context-synchronizing.

⁵ The alteration must not cause an implicit branch in real address space. Thus the real address of the context-altering instruction and of each subsequent instruction, up to and including the next context-synchronizing instruction, must be independent of whether the alteration has taken effect.

- ⁶ The elapsed time between the decremter reaching zero, or the transition of the selected time base bit for the fixed-interval or watchdog timer, and the signalling of the decremter, fixed-interval timer, or watchdog timer exception is not defined.
- ⁷ Synchronization requirements for changing any debug facility registers are implementation-dependent and are specified in the user's manual for the implementation.
- ⁸ For data accesses, the context-synchronizing instruction before the **tlbwe** or **tlbivax** instruction ensures that all accesses due to preceding instructions have completed to a point at which they have reported all exceptions they will cause. See Section 3.3.3.5.1, "Synchronization with tlbwe and tlbivax Instructions."
- ⁹ The context-synchronizing instruction after **tlbwe** or **tlbivax** ensures that subsequent accesses (data and instruction) use the updated value in the affected TLB entries. It does not ensure that all accesses previously translated by the TLB entries being updated have completed with respect to memory; if these completions must be ensured, **tlbwe** or **tlbivax** must be followed by an **msync** and by a context-synchronizing instruction. See Section 3.3.3.5.1, "Synchronization with tlbwe and tlbivax Instructions."

3.3.3.5.1 Synchronization with tlbwe and tlbivax Instructions

The following sequence shows why it is necessary, for data accesses, to ensure that all memory accesses due to instructions before the **tlbwe** or **tlbivax** have completed to a point at which they have reported all exceptions they cause. Assume that valid TLB entries exist for the target memory location when the sequence starts.

1. A program issues a load or store to a page.
2. The same program executes a **tlbwe** or **tlbivax** that invalidates the corresponding TLB entry.
3. The load or store instruction finally executes, and gets a TLB miss exception.

The TLB miss exception is semantically incorrect. To prevent it, a context-synchronizing instruction must be executed between steps 1 and 2.

3.3.3.5.2 Context Synchronization

An instruction or event is context synchronizing if it satisfies the requirements listed below. Context synchronizing operations include **sc**, **rfi**, **rfci**, and **rfmci**, and most interrupts.

1. The operation is not initiated or, in the case of **isync**, does not complete until all instructions already in execution have completed to a point at which they have reported all exceptions they cause.
2. The instructions that precede the operation complete execution in the context (including such parameters as privilege level, address space, and memory protection) in which they were initiated.
3. If the operation directly causes an interrupt (for example, **sc** directly causes a System Call interrupt) or is an interrupt, the operation is not initiated until no interrupt-causing exception exists having higher priority than the exception associated with the interrupt. See Section 5.11, "Exception Priorities."

4. The instructions that follow the operation are fetched and executed in the context established by the operation as required by the sequential execution model. (This requirement dictates that any prefetched instructions be discarded and that any effects and side effects of executing them speculatively may also be discarded, except as described in Section 11.1.3.1, “Memory Access Ordering.”)

As described in Section 3.3.3.5.3, “Execution Synchronization,” a context synchronizing operation is necessarily execution synchronizing. Unlike **msync** and **mbar**, such operations do not affect the order of memory accesses with respect to other mechanisms.

3.3.3.5.3 Execution Synchronization

An instruction is execution synchronizing if it satisfies items 1 and 2 of the definition of context synchronization (see Section 3.3.3.5.2, “Context Synchronization”). **msync** is treated like **isync** with respect to item 1 (that is, the conditions described in item 1 apply to completion of **msync**). Execution synchronizing instructions include **msync**, **mtmsr**, **wrtee**, and **wrteei**. All context synchronizing instructions are execution synchronizing.

Unlike a context synchronizing operation, an execution synchronizing instruction need not ensure that the instructions following it execute in the context established by that execution synchronizing instruction. This new context becomes effective sometime after the execution synchronizing instruction completes and before or at a subsequent context synchronizing operation.

3.3.3.5.4 Instruction-Related Interrupts

Interrupts are caused either directly by the execution of an instruction or by an asynchronous event. In either case, an exception may cause one of several types of interrupts to be invoked.

Examples of interrupts that can be caused directly by the execution of an instruction include but are not limited to the following:

- An attempt to execute a reserved-illegal instruction (illegal instruction exception type program interrupt).
- An attempt by an application program to execute a privileged instruction (privileged instruction exception type program interrupt).
- An attempt by an application program to access a privileged SPR (privileged instruction exception type program interrupt).
- An attempt by an application program to access an SPR that does not exist (unimplemented operation instruction exception type program interrupt).
- An attempt by a system program to access an SPR that does not exist (boundedly undefined).

- The execution of a defined instruction using an invalid form (illegal instruction exception type program interrupt, unimplemented operation exception type program interrupt, or privileged instruction exception type program interrupt).
- An attempt to access a memory location that is either unavailable (instruction TLB error interrupt or data TLB error interrupt) or not permitted (instruction storage interrupt or data storage interrupt).
- An attempt to access memory with an effective address alignment not supported by the implementation (alignment interrupt).
- The execution of a system call instruction (system call interrupt)
- The execution of a **trap** instruction whose trap condition is met (trap type program interrupt).
- The execution of a floating-point instruction when floating-point instructions are unavailable (floating-point unavailable interrupt) (not supported on the e500).
- The execution of a defined instruction that is not implemented by the implementation (illegal instruction exception or unimplemented operation exception type of program interrupt).
- The execution of an allocated instruction that is not implemented by the implementation (illegal instruction exception or unimplemented operation exception type of program interrupt).
- The execution of an allocated instruction when the auxiliary instruction is unavailable (auxiliary processor unavailable interrupt) (not supported on the e500).
- The execution of an allocated instruction that causes an auxiliary enabled exception (enabled exception type program interrupt).

APUs, such as the SPE, may define additional instruction-caused exceptions and interrupts. The invocation of an interrupt is precise, except that if one of the imprecise modes for invoking the floating-point enabled exception type program interrupt is in effect the invocation of the floating-point enabled exception type program interrupt may be imprecise. When the interrupt is invoked imprecisely, the excepting instruction does not appear to complete before the next instruction starts (because one of the effects of the excepting instruction, namely the invocation of the interrupt, has not yet occurred).

Chapter 5, “Interrupts and Exceptions,” describes interrupt conditions in detail.

3.4 Instruction Set Overview

This section provides a brief overview of the PowerPC instructions implemented in the e500 and highlights any special information with respect to how the e500 implements a particular instruction.

Note that some instructions have the following optional features:

- CR update—The dot (.) suffix on the mnemonic enables the update of the CR.
- Overflow option—The **o** suffix indicates that the overflow bit in the integer exception register (XER) is enabled.

3.4.1 Book E User-Level Instructions

This section discusses the user-level instructions defined in the Book E architecture.

3.4.1.1 Integer Instructions

This section describes the integer instructions. These consist of the following:

- Integer arithmetic instructions
- Integer compare instructions
- Integer logical instructions
- Integer rotate and shift instructions

Integer instructions use the content of the GPRs as source operands and place results into GPRs and the XER and CR fields.

3.4.1.1.1 Integer Arithmetic Instructions

Table 3-8 lists the integer arithmetic instructions for the PowerPC processors.

Table 3-8. Integer Arithmetic Instructions

Name	Mnemonic	Syntax
Add	add (add. addo addo.)	rD,rA,rB
Add Carrying	addc (addc. addco addco.)	rD,rA,rB
Add Extended	adde (adde. addeo addeo.)	rD,rA,rB
Add Immediate	addi	rD,rA,SIMM
Add Immediate Carrying	addic	rD,rA,SIMM
Add Immediate Carrying and Record	addic.	rD,rA,SIMM
Add Immediate Shifted	addis	rD,rA,SIMM
Add to Minus One Extended	addme (addme. addmeo addmeo.)	rD,rA
Add to Zero Extended	addze (addze. addzeo addzeo.)	rD,rA
Divide Word	divw (divw. divwo divwo.)	rD,rA,rB
Divide Word Unsigned	divwu divwu. divwuo divwuo.	rD,rA,rB
Multiply High Word	mulhw (mulhw.)	rD,rA,rB
Multiply High Word Unsigned	mulhwu (mulhwu.)	rD,rA,rB
Multiply Low Immediate	mulli	rD,rA,SIMM

Table 3-8. Integer Arithmetic Instructions (continued)

Name	Mnemonic	Syntax
Multiply Low Word	mullw (mullw. mullwo mullwo.)	rD,rA,rB
Negate	neg (neg. nego nego.)	rD,rA
Subtract From	subf (subf. subfo subfo.)	rD,rA,rB
Subtract from Carrying	subfc (subfc. subfco subfco.)	rD,rA,rB
Subtract from Extended	subfe (subfe. subfeo subfeo.)	rD,rA,rB
Subtract from Immediate Carrying	subfic	rD,rA,SIMM
Subtract from Minus One Extended	subfme (subfme. subfmeo subfmeo.)	rD,rA
Subtract from Zero Extended	subfze (subfze. subfzeo subfzeo.)	rD,rA

Although there is no subtract immediate instruction, its effect can be achieved by using an **addi** instruction with the immediate operand negated. Simplified mnemonics are provided that include this negation. Subtract instructions subtract the second operand (**rA**) from the third operand (**rB**). Simplified mnemonics are provided in which the third operand is subtracted from the second. See Appendix A, “Simplified Mnemonics,” for examples.

According to Book E, an implementation that executes instructions with the overflow exception enable bit (OE) set or that sets the carry bit (CA) can either execute these instructions slowly or prevent execution of the subsequent instruction until the operation completes. Chapter 4, “Execution Timing,” describes how the e500 handles CR dependencies. The summary overflow (SO) and overflow (OV) bits in the XER are set to reflect an overflow condition of a 32-bit result only if the instruction’s OE bit is set.

3.4.1.1.2 Integer Compare Instructions

The integer compare instructions algebraically or logically compare the contents of register **rA** with either the zero-extended value of the UIMM operand, the sign-extended value of the SIMM operand, or the contents of **rB**. The comparison is signed for the **cmpi** and **cmp** instructions, and unsigned for the **cmpli** and **cmpl** instructions. Table 3-9 summarizes the integer compare instructions. Note that the L bit must be 0 for 32-bit implementations.

Table 3-9. Integer 32-Bit Compare Instructions (L = 0)

Name	Mnemonic	Syntax
Compare	cmp	crfD,L,rA,rB
Compare Immediate	cmpi	crfD,L,rA,SIMM
Compare Logical	cmpl	crfD,L,rA,rB
Compare Logical Immediate	cmpli	crfD,L,rA,UIMM

The **crfD** operand can be omitted if the result of the comparison is to be placed in CR0. Otherwise the target CR field must be specified in **crfD** by using an explicit field number.

For information on simplified mnemonics for the integer compare instructions see Appendix A, “Simplified Mnemonics.”

3.4.1.1.3 Integer Logical Instructions

The logical instructions shown in Table 3-10 perform bit-parallel operations on the specified operands. Logical instructions with the CR updating enabled (uses dot suffix) and instructions **andi.** and **andis.** set CR field CR0 to characterize the result of the logical operation. Logical instructions do not affect XER[SO], XER[OV], or XER[CA].

See Appendix A, “Simplified Mnemonics,” for simplified mnemonic examples for integer logical operations.

Table 3-10. Integer Logical Instructions

Name	Mnemonic	Syntax	Implementation Notes
AND	and (and.)	rA,rS,rB	—
AND Immediate	andi.	rA,rS,UIMM	—
AND Immediate Shifted	andis.	rA,rS,UIMM	—
AND with Complement	andc (andc.)	rA,rS,rB	—
Count Leading Zeros Word	cntlzw (cntlzw.)	rA,rS	—
Equivalent	eqv (eqv.)	rA,rS,rB	—
Extend Sign Byte	extsb (extsb.)	rA,rS	—
Extend Sign Half Word	extsh (extsh.)	rA,rS	—
NAND	nand (nand.)	rA,rS,rB	—
NOR	nor (nor.)	rA,rS,rB	—
OR	or (or.)	rA,rS,rB	—
OR Immediate	ori	rA,rS,UIMM	Book E defines ori r0,r0,0 as the preferred form for a no-op instruction. The dispatcher may discard this instruction and only dispatch it to the completion queue but not to any execution unit.
OR Immediate Shifted	oris	rA,rS,UIMM	—
OR with Complement	orc (orc.)	rA,rS,rB	—
XOR	xor (xor.)	rA,rS,rB	—
XOR Immediate	xori	rA,rS,UIMM	—
XOR Immediate Shifted	xoris	rA,rS,UIMM	—

3.4.1.1.4 Integer Rotate and Shift Instructions

Rotation operations are performed on data from a GPR, and the result, or a portion of the result, is returned to a GPR. See Appendix A, “Simplified Mnemonics,” for a complete list of simplified mnemonics that allows simpler coding of often-used functions such as

clearing the left- or rightmost bits of a register, left or right justifying an arbitrary field, and simple rotates and shifts.

Integer rotate instructions rotate the contents of a register. The result of the rotation is either inserted into the target register under control of a mask (if a mask bit is set the associated bit of the rotated data is placed into the target register, and if the mask bit is cleared the associated bit in the target register is unchanged) or ANDed with a mask before being placed into the target register.

The integer rotate instructions are summarized in Table 3-11.

Table 3-11. Integer Rotate Instructions

Name	Mnemonic	Syntax
Rotate Left Word Immediate then AND with Mask	rlwinm (<i>rlwinm.</i>)	<i>rA,rS,SH,MB,ME</i>
Rotate Left Word then AND with Mask	rlwnm (<i>rlwnm.</i>)	<i>rA,rS,rB,MB,ME</i>
Rotate Left Word Immediate then Mask Insert	rlwimi (<i>rlwimi.</i>)	<i>rA,rS,SH,MB,ME</i>

The integer shift instructions (Table 3-12) perform left and right shifts. Immediate-form logical (unsigned) shift operations are obtained by specifying masks and shift values for certain rotate instructions. Simplified mnemonics (shown in Appendix A, “Simplified Mnemonics”) are provided to simplify coding of such shifts.

Multiple-precision shifts can be programmed as shown in Section B.2, “Multiple-Precision Shifts.” The integer shift instructions are summarized in Table 3-12.

Table 3-12. Integer Shift Instructions

Name	Mnemonic	Syntax
Shift Left Word	slw (<i>slw.</i>)	<i>rA,rS,rB</i>
Shift Right Word	srw (<i>srw.</i>)	<i>rA,rS,rB</i>
Shift Right Algebraic Word Immediate	srawi (<i>srawi.</i>)	<i>rA,rS,SH</i>
Shift Right Algebraic Word	sraw (<i>sraw.</i>)	<i>rA,rS,rB</i>

3.4.1.2 Load and Store Instructions

Load and store instructions are issued and translated in program order; however, the accesses can occur out of order. Synchronizing instructions are provided to enforce strict ordering. The e500 supports load and store instructions as follows:

- Integer load instructions
- Integer store instructions
- Integer load and store with byte-reverse instructions
- Integer load and store multiple instructions
- Memory synchronization instructions

- load and store instructions defined by the SPE APU for reading and writing the 64-bit GPRs. These are described in Section 3.11.1, “Signal Processing Engine (SPE) APU Instructions.”

The e500 does not implement the Book E floating-point load and store instructions.

Implementation Notes—The following describes how the e500 handles misalignment:

The e500 provides hardware support for misaligned memory accesses. It performs those accesses within a single cycle if the operand lies within a double-word boundary. Misaligned memory accesses that cross a double-word boundary degrade performance.

Although many misaligned memory accesses are supported in hardware, the frequent use of them is discouraged because they can compromise the overall performance of the processor. Only one outstanding misalignment at a time is supported, which means it is non-pipelined.

Accesses that cross a translation boundary can be restarted. That is, a misaligned access that crosses a page boundary is completely restarted if the second portion of the access causes a page fault. This can cause the first access to be repeated.

3.4.1.2.1 Self-Modifying Code

When a processor modifies a memory location that can be contained in the instruction cache, software must ensure that memory updates are visible to the fetching mechanism. This can be done by executing the following instruction sequence (using **dcbst** or **dcbf**):

dcbst (or dcbf)	update memory
msync	wait for update
icbi	remove (invalidate) copy in instruction cache
msync	ensure the ICBI invalidate is complete
isync	remove copy in own instruction buffer

These operations are required because the data cache is a write-back cache. Because instruction fetching bypasses the data cache, changes to items in the data cache cannot be reflected in memory until the fetch operations complete. The **msync** after the **icbi** is required to ensure that the **icbi** invalidation has completed in the instruction cache.

Special care must be taken to avoid coherency paradoxes in systems that implement unified secondary caches (like the e500), and designers should carefully follow the guidelines for maintaining cache coherency discussed in Chapter 12, “L1 Caches.”

3.4.1.2.2 Integer Load and Store Address Generation

Integer load and store operations generate effective addresses using register indirect with immediate index mode, register indirect with index mode, or register indirect mode, which are describes as follows:

- Register indirect with immediate index addressing for integer loads and stores

Instructions using this addressing mode contain a signed 16-bit immediate index (d operand) which is sign extended, and added to the contents of a general-purpose register specified in the instruction (rA operand) to generate the effective address. If the rA field of the instruction specifies r0, a value of zero is added to the immediate index (d operand) in place of the contents of r0. The option to specify rA or 0 is shown in the instruction descriptions as (rA|0).

Figure 3-1 shows how an effective address is generated when register indirect with immediate index addressing is used.

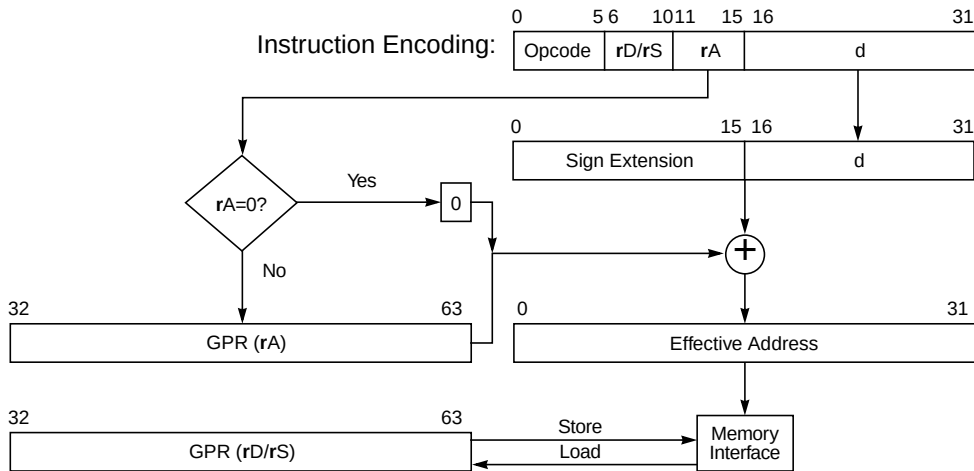


Figure 3-1. Register Indirect with Immediate Index Addressing for Integer Loads/Stores

- Register indirect with index addressing for integer loads and stores

Instructions using this addressing mode cause the contents of two general-purpose registers (specified as operands rA and rB) to be added in the generation of the effective address. A zero in place of the rA operand causes a zero to be added to the contents of the general-purpose register specified in operand rB. The option to specify rA or 0 is shown in the instruction descriptions as (rA|0).

Figure 3-2 shows how an effective address is generated when using register indirect with index addressing.

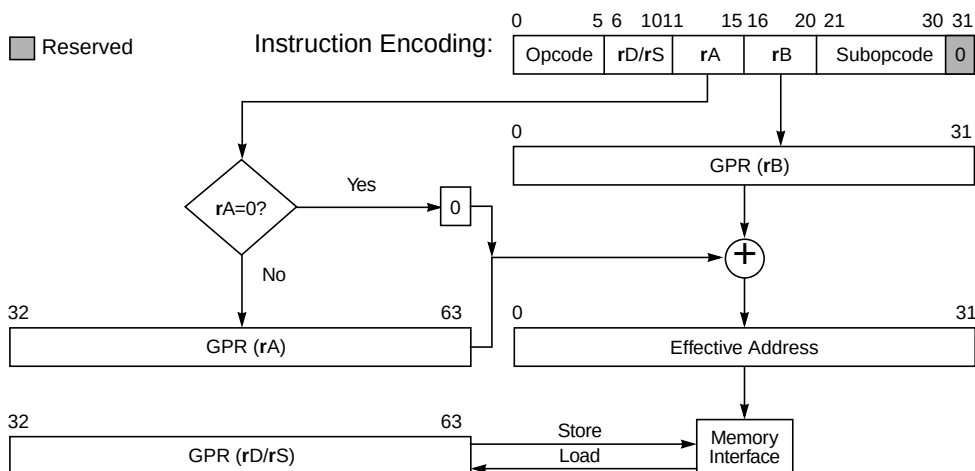


Figure 3-2. Register Indirect with Index Addressing for Integer Loads/stores

- Register Indirect Addressing for Integer Loads and Stores

Instructions using this addressing mode use the contents of the GPR specified by the **rA** operand as the effective address. A zero in the **rA** operand causes an effective address of zero to be generated. The option to specify **rA** or 0 is shown in the instruction descriptions as (**rA**|0).

Figure 3-3 shows how an effective address is generated when using register indirect addressing.

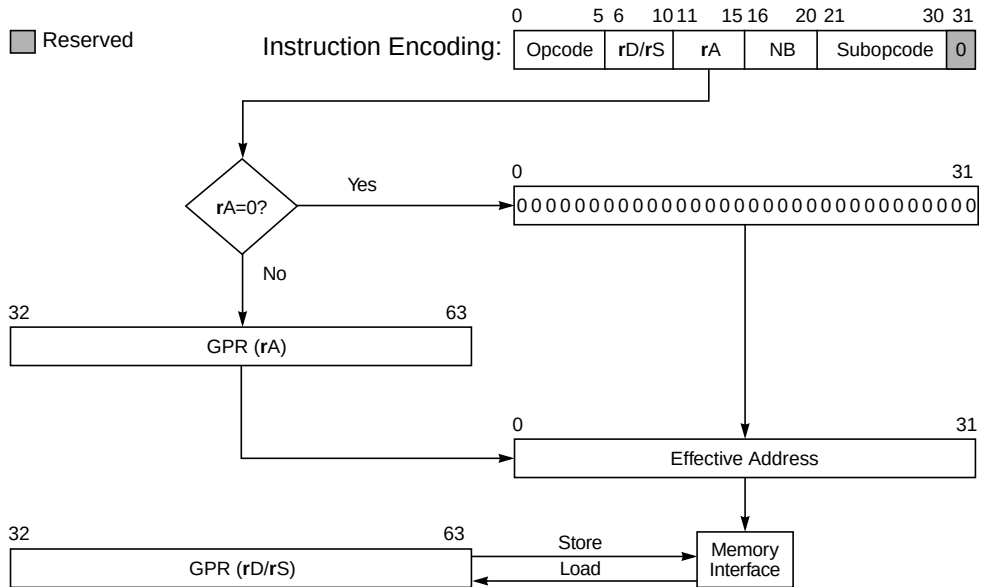


Figure 3-3. Register Indirect Addressing for Integer Loads/Stores

See Section 3.3.3.3, “Effective Address Calculation,” for information about calculating effective addresses. Note that in some implementations, operations that are not naturally aligned can suffer performance degradation. Section 5.7.6, “Alignment Interrupt,” for additional information about load and store address alignment interrupts.

3.4.1.2.3 Integer Load Instructions

Table 3-13 summarizes the integer load instructions.

Table 3-13. Integer Load Instructions

Name	Mnemonic	Syntax
Load Byte and Zero	lbz	rD,d(rA)
Load Byte and Zero Indexed	lbzx	rD,rA,rB
Load Byte and Zero with Update	lbzu	rD,d(rA)
Load Byte and Zero with Update Indexed	lbzux	rD,rA,rB
Load Half Word and Zero	lhz	rD,d(rA)
Load Half Word and Zero Indexed	lhzx	rD,rA,rB
Load Half Word and Zero with Update	lhzu	rD,d(rA)
Load Half Word and Zero with Update Indexed	lhzux	rD,rA,rB
Load Half Word Algebraic	lha	rD,d(rA)

Table 3-13. Integer Load Instructions (continued)

Name	Mnemonic	Syntax
Load Half Word Algebraic Indexed	lhax	rD,rA,rB
Load Half Word Algebraic with Update	lhau	rD,d(rA)
Load Half Word Algebraic with Update Indexed	lhaux	rD,rA,rB
Load Word and Zero	lwz	rD,d(rA)
Load Word and Zero Indexed	lwzx	rD,rA,rB
Load Word and Zero with Update	lwzu	rD,d(rA)
Load Word and Zero with Update Indexed	lwzux	rD,rA,rB

The following notes describe the e500 implementation of integer load instructions:

- Book E cautions programmers that some implementations of the architecture can execute the load half algebraic (**lha**, **lhax**) instructions with greater latency than other types of load instructions. This is not the case for the e500; these instructions operate with the same latency as other load instructions.
- Book E cautions programmers that some implementations of the architecture can run the load/store byte-reverse (**lhbrrx**, **lbrx**, **sthbrrx**, **stwbrrx**) instructions with greater latency than other types of load/store instructions. This is not the case for the e500. These instructions operate with the same latency as the other load/store instructions.
- The Book E architecture defines **lwarx** and **stwcx**. as a way to update memory atomically. In the e500, reservations are made on behalf of aligned 32-byte sections of the memory address space. Executing **lwarx** and **stwcx**. to a page marked write-through causes a data storage interrupt if the page is marked cacheable write-through (WIM = 10x); but as with other memory accesses, data storage interrupts can result for other reasons such as a protection violations or page faults.

3.4.1.2.4 Integer Store Instructions

For integer store instructions, the contents of **rS** are stored into the byte, half word, word or double word in memory addressed by the EA (effective address). Many store instructions have an update form, in which **rA** is updated with the EA. For these forms, the following rules apply:

- If **rA** $\neq$ 0, the effective address is placed into **rA**.
- If **rS** = **rA**, the contents of register **rS** are copied to the target memory element and the generated EA is placed into **rA** (**rS**).

The Book E architecture defines store with update instructions with **rA** = 0 as an invalid form. In addition, it defines integer store instructions with the CR update option enabled

(Rc field, bit 31, in the instruction encoding = 1) to be an invalid form. Table 3-14 summarizes the integer store instructions.

Table 3-14. Integer Store Instructions

Name	Mnemonic	Syntax
Store Byte	stb	rS,d(rA)
Store Byte Indexed	stbx	rS,rA,rB
Store Byte with Update	stbu	rS,d(rA)
Store Byte with Update Indexed	stbux	rS,rA,rB
Store Half Word	sth	rS,d(rA)
Store Half Word Indexed	sthx	rS,rA,rB
Store Half Word with Update	sthbu	rS,d(rA)
Store Half Word with Update Indexed	sthux	rS,rA,rB
Store Word	stw	rS,d(rA)
Store Word Indexed	stwx	rS,rA,rB
Store Word with Update	stwu	rS,d(rA)
Store Word with Update Indexed	stwux	rS,rA,rB

3.4.1.2.5 Integer Load and Store with Byte-Reverse Instructions

Table 3-15 describes integer load and store with byte-reverse instructions. These books were defined in part to support the original PowerPC definition of little-endian byte ordering. Note that Book E supports true little endian on a per-page basis. For more information, see Section 3.3.3.4, “Byte Ordering.”

Table 3-15. Integer Load and Store with Byte-Reverse Instructions

Name	Mnemonic	Syntax
Load Half Word Byte-Reverse Indexed	lhbrx	rD,rA,rB
Load Word Byte-Reverse Indexed	lwbrx	rD,rA,rB
Store Half Word Byte-Reverse Indexed	sthbrx	rS,rA,rB
Store Word Byte-Reverse Indexed	stwbrx	rS,rA,rB

3.4.1.2.6 Integer Load and Store Multiple Instructions

The load/store multiple instructions are used to move blocks of data to and from the GPRs. The load multiple and store multiple instructions can have operands that require memory accesses crossing a 4-Kbyte page boundary. As a result, these instructions can be interrupted by a data storage interrupt associated with the address translation of the second page. Note that if one of these instructions is interrupted, it may be restarted, requiring multiple memory accesses.

The Book E architecture defines the Load Multiple Word (**lmw**) instruction (with **rA** in the range of registers to be loaded as an invalid form. Load and store multiple accesses must be word aligned; otherwise, they cause an alignment exception.

Table 3-16. Integer Load and Store Multiple Instructions

Name	Mnemonic	Syntax
Load Multiple Word	lmw	rD,d(rA)
Store Multiple Word	stmw	rS,d(rA)

3.4.1.3 Branch and Flow Control Instructions

Some branch instructions can redirect instruction execution conditionally based on the value of bits in the CR.

3.4.1.3.1 Branch Instruction Address Calculation

Branch instructions can alter the sequence of instruction execution. Instruction addresses are always assumed to be word aligned; the Book E processors ignore the two low-order bits of the generated branch target address.

Branch instructions compute the EA of the next instruction address using the following addressing modes:

- Branch relative
- Branch conditional to relative address
- Branch to absolute address
- Branch conditional to absolute address
- Branch conditional to link register
- Branch conditional to count register

3.4.1.3.2 Branch Relative Addressing Mode

Instructions that use branch relative addressing generate the next instruction address by sign extending and appending 0b00 to the immediate displacement operand LI, and adding the resultant value to the current instruction address. Branches using this addressing mode have the absolute addressing option disabled (AA field, bit 30, in the instruction encoding = 0). The link register (LR) update option can be enabled (LK field, bit 31, in the instruction encoding = 1). This option causes the effective address of the instruction following the branch instruction to be placed in the LR.

Figure 3-4 shows how the branch target address is generated when using the branch relative addressing mode.

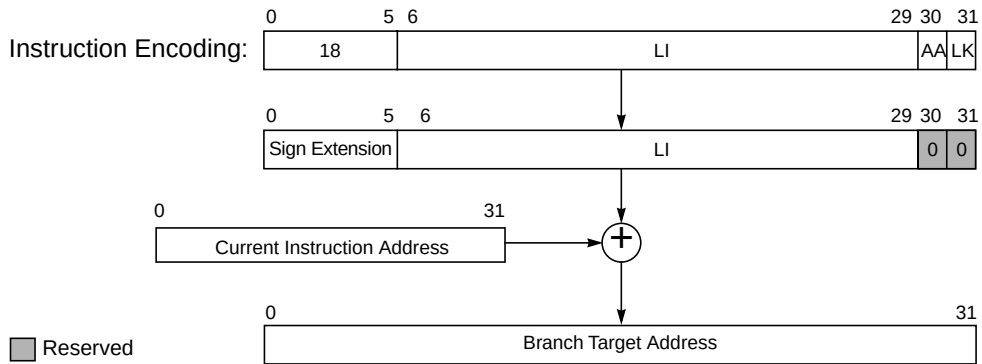


Figure 3-4. Branch Relative Addressing

3.4.1.3.3 Branch Conditional to Relative Addressing Mode

If the branch conditions are met, instructions that use the branch conditional to relative addressing mode generate the next instruction address by sign extending and appending 0b00 to the immediate displacement operand (BD) and adding the resultant value to the current instruction address. Branches using this addressing mode have the absolute addressing option disabled (AA field, bit 30, in the instruction encoding = 0). The link register update option can be enabled (LK field, bit 31, in the instruction encoding = 1). This option causes the effective address of the instruction following the branch instruction to be placed in the LR.

Figure 3-5 shows how the branch target address is generated when using the branch conditional relative addressing mode.

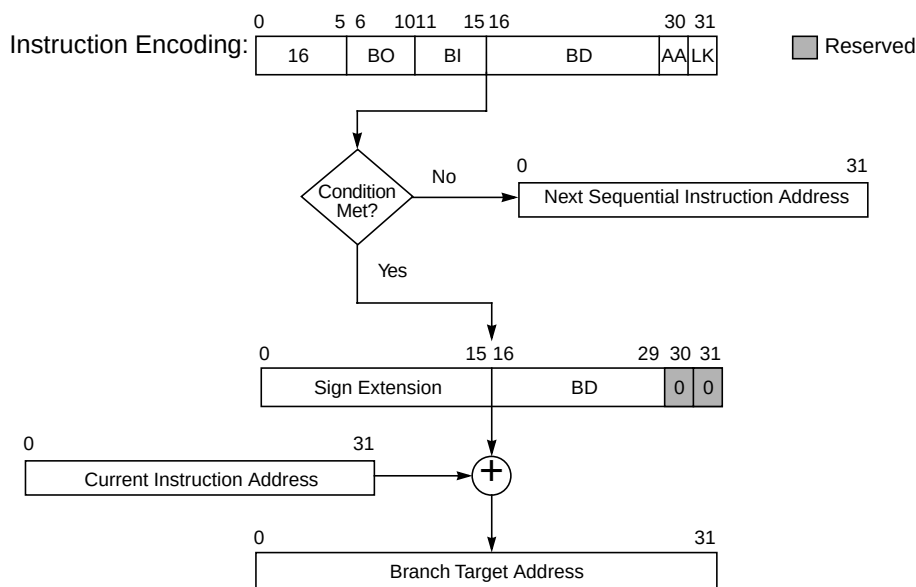


Figure 3-5. Branch Conditional Relative Addressing

3.4.1.3.4 Branch to Absolute Addressing Mode

Instructions that use branch to absolute addressing mode generate the next instruction address by sign extending and appending 0b00 to the LI operand. Branches using this addressing mode have the absolute addressing option enabled (AA field, bit 30, in the instruction encoding = 1). The link register update option can be enabled (LK field, bit 31, in the instruction encoding = 1). This option causes the effective address of the instruction following the branch instruction to be placed in the LR.

Figure 3-6 shows how the branch target address is generated when using the branch to absolute addressing mode.

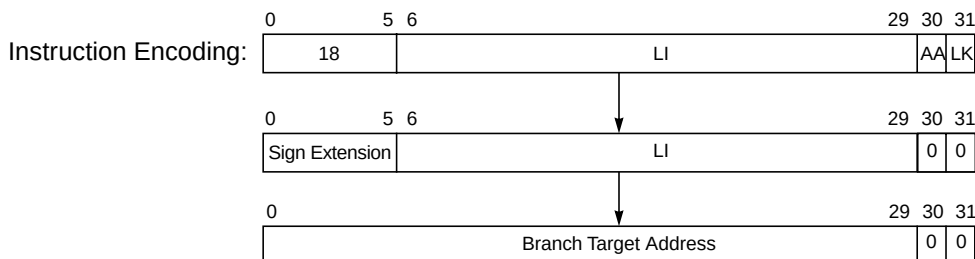


Figure 3-6. Branch to Absolute Addressing

3.4.1.3.5 Branch Conditional to Absolute Addressing Mode

If the branch conditions are met, instructions that use the branch conditional to absolute addressing mode generate the next instruction address by sign extending and appending 0b00 to the BD operand. Branches using this addressing mode have the absolute addressing mode option enabled (AA field, bit 30, in the instruction encoding = 1). The link register update option can be enabled (bit 31 (LK) in the instruction encoding = 1). This option causes the effective address of the instruction following the branch instruction to be placed in the LR.

Figure 3-7 shows how the branch target address is generated when using the branch conditional to absolute addressing mode.

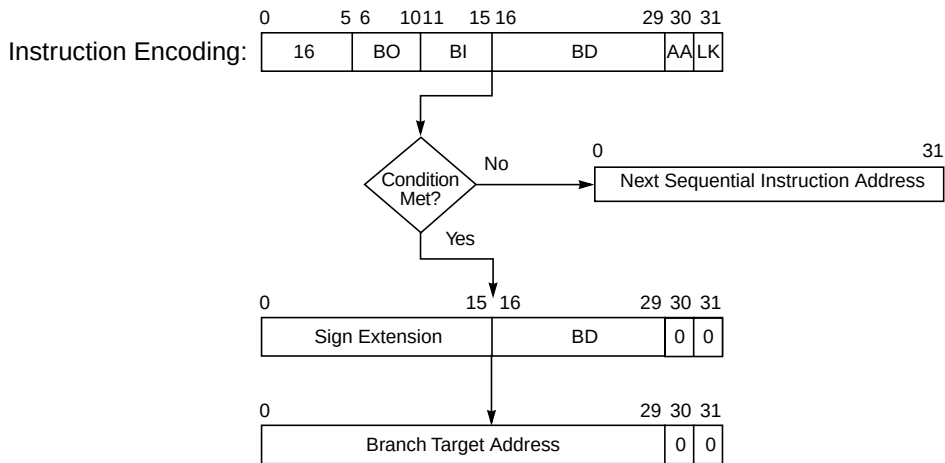


Figure 3-7. Branch Conditional to Absolute Addressing

3.4.1.3.6 Branch Conditional to Link Register Addressing Mode

If the branch conditions are met, the branch conditional to link register instruction generates the next instruction address by fetching the contents of the LR and clearing the two low-order bits to zero. The link register update option can be enabled (LK field, bit 31, in the instruction encoding = 1). This option causes the effective address of the instruction following the branch instruction to be placed in the LR.

Figure 3-8 shows how the branch target address is generated when using the branch conditional to link register addressing mode.

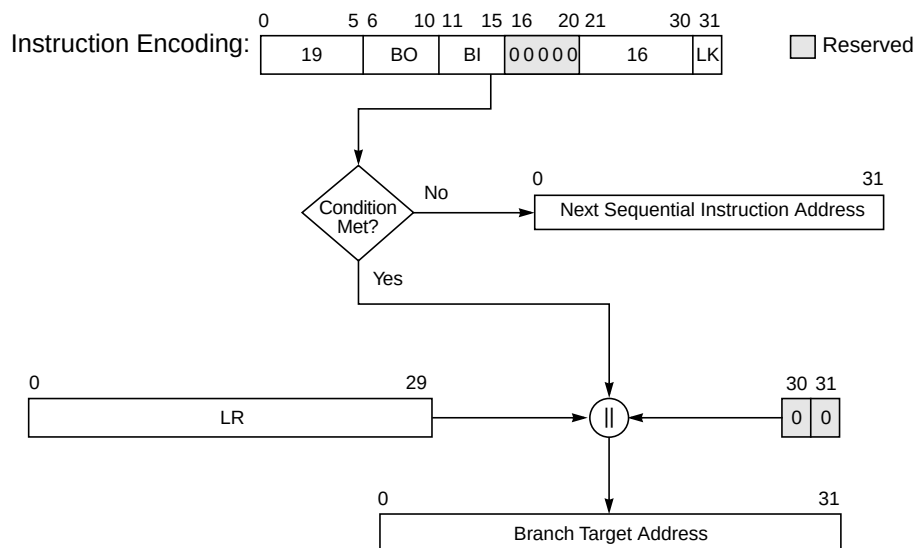


Figure 3-8. Branch Conditional to Link Register Addressing

3.4.1.3.7 Branch Conditional to Count Register Addressing Mode

If the branch conditions are met, the branch conditional to count register instruction generates the next instruction address by fetching the contents of the count register (CTR) and clearing the two low-order bits to zero. The link register update option can be enabled (LK field, bit 31, in the instruction encoding = 1). This option causes the effective address of the instruction following the branch instruction to be placed in the LR.

Figure 3-9 shows how the branch target address is generated when using the branch conditional to count register addressing mode.

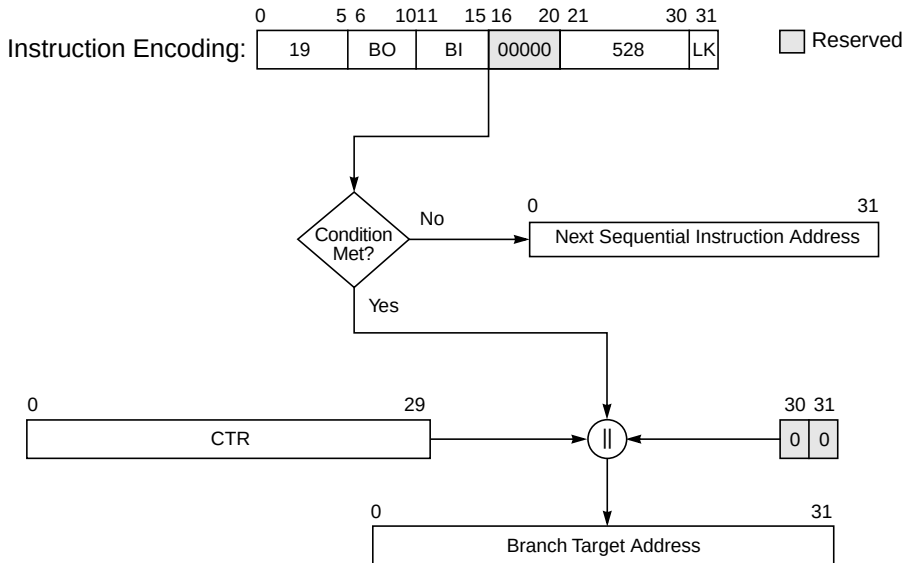


Figure 3-9. Branch Conditional to Count Register Addressing

3.4.1.4 Conditional Branch Control

NOTE

The e500 does not implement the static branch prediction defined in Book E and described here. On the e500, the BO operand is ignored for branch prediction. The e500 instead implements dynamic branch prediction as part of the branch table buffer (BTB), described in Section 4.4.1, “Branch Unit Execution Timing.”

For branch conditional instructions, the BO operand specifies the conditions under which the branch is taken. The first four bits of the BO operand specify how the branch is affected by or affects the condition and count registers. The fifth bit, shown in Table 3-17 as having the value *y*, is used by some implementations for branch prediction as described below.

Figure 3-10. BO Bit Descriptions

BO Bits	Description
0	Setting this bit causes the CR bit to be ignored
1	Bit value to test against
2	Setting this causes the decrement to not be decremented

Figure 3-10. BO Bit Descriptions (continued)

BO Bits	Description
3	Setting this bit reverses the sense of the CTR test.
4	Used for the y bit, which provides a hint about whether a conditional branch is likely to be taken and may be used by some implementations to improve performance.

The encodings for the BO operands are shown in Table 3-17

Table 3-17. BO Operand Encodings

BO	Description
0000y	Decrement the CTR, then branch if the decremented CTR $\neq$ 0 and the condition is FALSE.
0001y	Decrement the CTR, then branch if the decremented CTR = 0 and the condition is FALSE.
001zy	Branch if the condition is FALSE.
0100y	Decrement the CTR, then branch if the decremented CTR $\neq$ 0 and the condition is TRUE.
0101y	Decrement the CTR, then branch if the decremented CTR = 0 and the condition is TRUE.
011zy	Branch if the condition is TRUE.
1z00y	Decrement the CTR, then branch if the decremented CTR $\neq$ 0.
1z01y	Decrement the CTR, then branch if the decremented CTR = 0.
1z1zz	Branch always.

In this table, z indicates a bit that is ignored. Note that the z bits should be cleared, as they may be assigned a meaning in some future version of the architecture.

The y bit provides a hint about whether a conditional branch is likely to be taken and may be used by some implementations to improve performance.

The branch always encoding of the BO operand does not have a y bit.

Clearing the y bit indicates a predicted behavior for the branch instruction as follows:

- For **bcx** with a negative value in the displacement operand, the branch is taken.
- In all other cases (**bcx** with a non-negative value in the displacement operand, **bclrx**, or **bcctrx**), the branch is not taken.

Setting the y bit reverses the preceding indications.

The sign of the displacement operand is used as described above even if the target is an absolute address. The default value for the y bit should be 0 and should be set to 1 only if software has determined that the prediction corresponding to y = 1 is more likely to be correct than the prediction corresponding to y = 0. Software that does not compute branch predictions should clear the y bit.

In most cases, the branch should be predicted to be taken if the value of the following expression is 1, and predicted to fall through if the value is 0.

$$((\text{BO}[0] \ \& \ \text{BO}[2]) \mid \text{S}) \approx \text{BO}[4]$$

In the expression above, S (bit 16 of the branch conditional instruction coding) is the sign bit of the displacement operand if the instruction has a displacement operand and is 0 if the operand is reserved. BO[4] is the y bit, or 0 for the branch always encoding of the BO operand. (Advantage is taken of the fact that, for **bclrx** and **bcctrx**, bit 16 of the instruction is part of a reserved operand and therefore must be 0.)

The 5-bit BI operand in branch conditional instructions specifies which of the 32 bits in the CR represents the condition to test. The CR bit selected is the BI + 32 as shown in Table 2-5.

When the branch instructions contain immediate addressing operands, the target addresses can be computed sufficiently ahead of the branch instruction that instructions can be fetched along the target path. If the branch instructions use the link and count registers, instructions along the target path can be fetched if the link or count register is loaded sufficiently ahead of the branch instruction.

Branching can be conditional or unconditional, and optionally a branch return address is created by storing the effective address of the instruction following the branch instruction in the LR after the branch target address has been computed. This is done regardless of whether the branch is taken.

3.4.1.4.1 Branch Instructions

Table 3-18 lists the branch instructions provided by the Book E processors. A set of simplified mnemonics and symbols is provided for the most frequently used forms of branch conditional, compare, trap, rotate and shift, and certain other instructions. See Appendix A, “Simplified Mnemonics.” Note that the e500 does not use the BO operand for static branch prediction.

Table 3-18. Branch Instructions

Name	Mnemonic	Syntax
Branch	b (ba bl bla)	target_addr
Branch Conditional	bc (bca bcl bcla)	BO, BI, target_addr
Branch Conditional to Link Register	bclr (bclrl)	BO, BI
Branch Conditional to Count Register	bcctr (bcctrl)	BO, BI

Note that the e500 implements the Instruction Select instruction, **isel**, which can be used to more efficiently handle sequences with multiple conditional branches. Its syntax is given in Section 3.11.3, “Instruction Select (isel) APU.” A detailed description including an example of how **isel** can be used can be found in Section 10.1.2, “Using isel to Improve Conditional Branch Performance.”

3.4.1.4.2 Condition Register Logical Instructions

Condition register logical instructions, shown in Table 3-19, and the Move Condition Register Field (**mcrf**) instruction are also defined as flow control instructions.

Table 3-19. Condition Register Logical Instructions

Name	Mnemonic	Syntax
Condition Register AND	crand	crbD,crbA,crbB
Condition Register OR	cror	crbD,crbA,crbB
Condition Register XOR	crxor	crbD,crbA,crbB
Condition Register NAND	crnand	crbD,crbA,crbB
Condition Register NOR	crnor	crbD,crbA,crbB
Condition Register Equivalent	creqv	crbD,crbA,crbB
Condition Register AND with Complement	crandc	crbD,crbA,crbB
Condition Register OR with Complement	crorc	crbD,crbA,crbB
Move Condition Register Field	mcrf	crfD,crfS

Note that if the LR update option is enabled for any of these instructions, the Book E architecture defines these forms of the instructions as invalid.

3.4.1.4.3 Trap Instructions

The trap instructions shown in Table 3-20 test for a specified set of conditions. If any of the conditions tested by a trap instruction are met, the system trap type program interrupt is taken. For more information, see Section 5.7.7, “Program Interrupt.” If the tested conditions are not met, instruction execution continues normally.

Table 3-20. Trap Instructions

Name	Mnemonic	Syntax
Trap Word Immediate	twi	TO,rA,SIMM
Trap Word	tw	TO,rA,rB

See Appendix A, “Simplified Mnemonics.”

3.4.1.5 System Linkage Instruction

The System Call (**sc**) instruction permits a program to call on the system to perform a service; see Table 3-21 and Section 3.4.2.1, “System Linkage Instructions.”

Table 3-21. System Linkage Instruction

Name	Mnemonic	Syntax
System Call	sc	—

Executing this instruction causes the system call interrupt handler to be invoked. For more information, see Section 5.7.8, “System Call Interrupt.”

3.4.1.6 Processor Control Instructions

Processor control instructions are used to read from and write to the condition register (CR), machine state register (MSR), and special-purpose registers (SPRs).

3.4.1.6.1 Move to/from Condition Register Instructions

Table 3-22 summarizes the instructions for reading from or writing to the condition register.

Table 3-22. Move to/from Condition Register Instructions

Name	Mnemonic	Syntax
Move to Condition Register Fields	mtrcf	CRM,rS
Move to Condition Register from XER	mcrxr	crfD
Move from Condition Register	mfcrr	rD

Implementation Note—The Book E architecture states that the move to condition register fields (**mtrcf**) instruction can perform more slowly when only a portion of the fields are updated as opposed to all the fields. This is not the case for the e500.

3.4.1.6.2 Move To/From Special-Purpose Register Instructions

Table 3-23 lists the **mtspr** and **mfspir** instructions.

Table 3-23. Move to/from Special-Purpose Register Instructions

Name	Mnemonic	Syntax
Move to Special-Purpose Register	mtspr	SPR,rS
Move from Special-Purpose Register	mfspir	rD,SPR

Table 3-24 summarizes all SPRs defined in Book E, indicating which are user-level access. The column ‘SPRN’ (SPR number) lists register numbers that are used in the instruction mnemonics.

Table 3-24. Book E Special-Purpose Registers (by SPR Abbreviation)

SPR Abbreviation	Name	Defined SPR Number		Access	Supervisor Only	Section/ Page
		Decimal	Binary			
CSRR0	Critical save/restore register 0	58	00001 11010	Read/Write	Yes	2.7.1.3/2-24
CSRR1	Critical save/restore register 1	59	00001 11011	Read/Write	Yes	2.7.1.4/2-24
CTR	Count register	9	00000 01001	Read/Write	No	2.4.3/2-14

Table 3-24. Book E Special-Purpose Registers (by SPR Abbreviation) (continued)

SPR Abbreviation	Name	Defined SPR Number		Access	Supervisor Only	Section/ Page
		Decimal	Binary			
DAC1	Data address compare 1	316	01001 11100	Read/Write	Yes	2.13.4/2-56
DAC2	Data address compare 2	317	01001 11101			
DBCR0	Debug control register 0	308	01001 10100	Read/Write	Yes	2.13.1/2-50
DBCR1	Debug control register 1	309	01001 10101	Read/Write	Yes	
DBCR2	Debug control register 2	310	01001 10110	Read/Write	Yes	
DBSR	Debug status register	304	01001 10000	Read/Clear ¹	Yes	2.13.2/2-54
DEAR	Data exception address register	61	00001 11101	Read/Write	Yes	2.7.1.5/2-25
DEC	Decrementer	22	00000 10110	Read/Write	Yes	2.6.4/2-22
DECAR	Decrementer auto-reload	54	00001 10110	Write-only		
ESR	Exception syndrome register	62	00001 11110	Read/Write	Yes	2.7.1.8/2-26
IAC1	Instruction address compare 1	312	01001 11000	Read/Write	Yes	2.13.3/2-56
IAC2	Instruction address compare 2	313	01001 11001			
IVOR0	Critical input	400	01100 10000	Read/Write	Yes	2.7.1.7/2-25
IVOR1	— interrupt offset	401	01100 10001			
IVOR2	Data storage interrupt offset	402	01100 10010			
IVOR3	Instruction storage interrupt offset	403	01100 10011			
IVOR4	External input interrupt offset	404	01100 10100			
IVOR5	Alignment interrupt offset	405	01100 10101			
IVOR6	Program interrupt offset	406	01100 10110			
IVOR7	Floating-point unavailable interrupt offset (not supported on the e500)	407	01100 10111			
IVOR8	System call interrupt offset	408	01100 11000			
IVOR9	Auxiliary processor unavailable interrupt offset (not supported on the e500)	409	01100 11001			
IVOR10	Decrementer interrupt offset	410	01100 11010			
IVOR11	Fixed-interval timer interrupt offset	411	01100 11011			
IVOR12	Watchdog timer interrupt offset	412	01100 11100			
IVOR13	Data TLB error interrupt offset	413	01100 11101			
IVOR14	Instruction TLB error interrupt offset	414	01100 11110			
IVOR15	Debug interrupt offset	415	01100 11111			
IVPR	Interrupt vector	63	00001 11111	Read/Write	Yes	2.7.1.6/2-25
LR	Link register	8	00000 01000	Read/Write	No	2.4.2/2-13

Table 3-24. Book E Special-Purpose Registers (by SPR Abbreviation) (continued)

SPR Abbreviation	Name	Defined SPR Number		Access	Supervisor Only	Section/ Page
		Decimal	Binary			
PID	Process ID register ²	48	00001 10000	Read/Write	Yes	2.12.1/2-41
PIR	Processor ID register	286	01000 11110	Read-only	Yes	2.5.2/2-16
PVR	Processor version register	287	01000 11111	Read-only	Yes	2.5.3/2-17
SPRG0	SPR general 0	272	01000 10000	Read/Write	Yes	2.8/2-31
SPRG1	SPR general 1	273	01000 10001	Read/Write	Yes	
SPRG2	SPR general 2	274	01000 10010	Read/Write	Yes	
SPRG3	SPR general 3	259	01000 00011	Read-only	No ³	
		275	01000 10011	Read/Write	Yes	
SPRG4	SPR general 4	260	01000 00100	Read-only	No	
		276	01000 10100	Read/Write	Yes	
SPRG5	SPR general 5	261	01000 00101	Read-only	No	
		277	01000 10101	Read/Write	Yes	
SPRG6	SPR general 6	262	01000 00110	Read-only	No	
		278	01000 10110	Read/Write	Yes	
SPRG7	SPR general 7	263	01000 00111	Read-only	No	
		279	01000 10111	Read/Write	Yes	
SRR0	Save/restore register 0	26	00000 11010	Read/Write	Yes	2.7.1.1/2-23
SRR1	Save/restore register 1	27	00000 11011	Read/Write	Yes	2.7.1.2/2-23
TBL	Time base lower	268	01000 01100	Read-only	No	2.6.3/2-21
		284	01000 11100	Write-only	Yes	
TBU	Time base upper	269	01000 01101	Read-only	No	
		285	01000 11101	Write-only	Yes	
TCR	Timer control register	340	01010 10100	Read/Write	Yes	2.6.1/2-19
TSR	Timer status register	336	01010 10000	Read/Clear ⁴	Yes	2.6.2/2-20
USPRG0	User SPR general 0 ⁵	256	01000 00000	Read/Write	No	2.8/2-31
XER	Integer exception register	1	00000 00001	Read/Write	No	2.3.2/2-9

¹ The DBSR is read using **mfsprr**. It cannot be directly written to. Instead, DBSR bits corresponding to 1 bits in the GPR can be cleared using **mtsprr**.

² Implementations may support more than one PID. The e500 implements the Book E–defined PID as PID0.

³ User-mode read access to SPRG3 is implementation-dependent.

⁴ The TSR is read using **mfsprr**. It cannot be directly written to. Instead, TSR bits corresponding to 1 bits in the GPR can be cleared using **mtsprr**.

⁵ USPRG0 is a separate physical register from SPRG0.

Table 3-25 lists implementation-specific SPRs of the core complex, indicating which can be accessed by user-level software. Compilers should recognize the mnemonic name given in Table A-23 when parsing instructions.

Table 3-25. Implementation-Specific SPRs (by SPR Abbreviation)

SPR Abbreviation	Name	SPR Number	Access	Supervisor Only	Section/ Page
BBEAR	Branch buffer entry address register	513	Read/Write	No	2.9.1/2-32
BBTAR	Branch buffer target address register	514	Read/Write	No	2.9.2/2-33
BUCSR	Branch unit control and status register	1013	Read/Write	Yes	2.9.3/2-33
HID0	Hardware implementation dependent reg 0	1008	Read/Write	Yes	2.10.1/2-35
HID1	Hardware implementation dependent reg 1	1009	Read/Write	Yes	2.10.1/2-35
IARR	Instruction address resume register	518	Read/Write	No	2.11.3/2-39
IVOR32	SPE APU unavailable interrupt offset	528	Read/Write	Yes	2.7.1.7/2-25
IVOR33	SPE floating-point data exception interrupt offset	529	Read/Write	Yes	
IVOR34	SPE floating-point round exception interrupt offset	530	Read/Write	Yes	
IVOR35	Performance monitor	531	Read/Write	Yes	
L1CFG0	L1 cache configuration register 0	515	Read-only	No	2.11.3/2-39
L1CFG1	L1 cache configuration register 1	516	Read-only	No	2.11.4/2-40
L1CSR0	L1 cache control and status register 0	1010	Read/Write	Yes	2.11.1/2-37
L1CSR1	L1 cache control and status register 1	1011	Read/Write	Yes	2.11.2/2-38
MAS0	MMU assist register 0	624	Read/Write	Yes	2.12.5.1/2-45
MAS1	MMU assist register 1	625	Read/Write	Yes	2.12.5.2/2-46
MAS2	MMU assist register 2	626	Read/Write	Yes	2.12.5.3/2-46
MAS3	MMU assist register 3	627	Read/Write	Yes	2.12.5.4/2-48
MAS4	MMU assist register 4	628	Read/Write	Yes	2.12.5.5/2-48
MAS5	MMU assist register 5. Defined by Motorola Book E implementation standard but not implemented in the e500.	629	Read/Write	Yes	—
MAS6	MMU assist register 6	630	Read/Write	Yes	2.12.5.6/2-49
MCAR	Machine check address register	573	Read-only	Yes	2.7.2.3/2-29
MCSR	Machine check syndrome register	572	Read/Write	Yes	2.7.2.4/2-29
MCSRRO	Machine-check save/restore register 0	570	Read/Write	Yes	2.7.2.1/2-28
MCSRRI	Machine-check save/restore register 1	571	Read/Write	Yes	2.7.2.2/2-28
MMUCFG	MMU configuration register	1015	Read-only	Yes	2.12.3/2-42
MMUCSR0	MMU control and status register 0	1012	Read/Write	Yes	2.12.2/2-41
NPIDR ¹	Nexus processor ID register	517	Read/Write	No	2.17/2-66

Table 3-25. Implementation-Specific SPRs (by SPR Abbreviation) (continued)

SPR Abbreviation	Name	SPR Number	Access	Supervisor Only	Section/ Page
PID0	Process ID register 0. Book E defines only this PID register and refers to as PID rather than PID0.	48	Read/Write	Yes	2.12.1/2-41
PID1	Process ID register 1	633	Read/Write	Yes	
PID2	Process ID register 2	634	Read/Write	Yes	
SPEFSCR	Signal processing and embedded floating-point status and control register	512	Read/Write	No	2.14.1/2-57
TLB0CFG	TLB configuration register 0	688	Read-only	Yes	2.12.4/2-43
TLB1CFG	TLB configuration register 1	689	Read-only	Yes	2.12.4.2/2-44

¹ NPIDR is transferred to the NEXUS port whenever it is written

3.4.1.7 Memory Synchronization Instructions

Memory synchronization instructions control the order in which memory operations complete with respect to asynchronous events and the order in which memory operations are seen by other mechanisms that access memory. See Section 3.4.1.8, “Atomic Update Primitives Using `lwarx` and `stwcx.`,” for additional information about these instructions and about related aspects of memory synchronization. See Table 3-26 for a summary.

Table 3-26. Memory Synchronization Instructions

Name	Mnemonic	Syntax	Implementation Notes
Load Word and Reserve Indexed	<code>lwarx</code>	<code>rD,rA,rB</code>	<code>lwarx</code> with <code>stwcx.</code> can emulate semaphore operations such as test and set, compare and swap, exchange memory, and fetch and add. Both instructions must use the same EA. Reservation granularity is implementation-dependent. The e500 makes reservations on behalf of aligned 32-byte sections of address space. Executing <code>lwarx</code> and <code>stwcx.</code> to a page marked write-through (WIMG = 10xx) or when the data cache is locked causes a data storage interrupt. If the location is not word-aligned, an alignment interrupt occurs. See Section 3.4.1.8, “Atomic Update Primitives Using <code>lwarx</code> and <code>stwcx.</code> ”
Store Word Conditional Indexed	<code>stwcx.</code>	<code>rS,rA,rB</code>	
Instruction Synchronize	<code>isync</code>	—	<code>isync</code> is refetch serializing; the e500 must wait for previous instructions to complete before <code>isync</code> executes, which purges all instructions from the processor and refetches the next instruction. <code>isync</code> is not executed until all previous instructions cannot cause an interrupt. <code>isync</code> does not wait for pending stores in the store queue to complete. Any subsequent instruction sees all effects of instructions before the <code>isync</code> . Because it prevents execution of subsequent instructions until preceding instructions complete, if an <code>isync</code> follows a conditional branch instruction that depends on the value returned by a preceding load, the load on which the branch depends is performed before any loads caused by instructions after the <code>isync</code> even if the effects of the dependency are independent of the value loaded (for example, the value is compared to itself and the branch tests selected, <code>CRn[EQ]</code>), and even if the branch target is the next sequential instruction to be executed.

Table 3-26. Memory Synchronization Instructions (continued)

Name	Mnemonic	Syntax	Implementation Notes
Memory Barrier	mbar	MO	mbar provides a pipelined memory barrier. (Note that mbar uses the same opcode as eiemo, which is not defined by Book E.) The e500 implements two variations of mbar, depending on the MO field (bits 6–10) of the instruction. MO=0 (or any value other than 1 on the e500), mbar behaves identically to msync. MO=1, mbar is a weaker, faster memory barrier; it ensures that all data memory accesses caused by previous instructions have completed before any accesses caused by subsequent instructions. This order is seen by all mechanisms. However, unlike msync and mbar with MO=0, mbar does not wait for its address tenure to be performed successfully on the bus before allowing subsequent instructions to complete. This provides a faster way to order data accesses. The following sequence shows one use of mbar in supporting shared data, ensuring the action is completed before the lock is released. <pre> P1 P2 lock . . . read & write . . . mbar . . . free lock lock . . . read & write . . . mbar . . . free lock </pre>
Memory Synchronize	msync	—	msync provides a memory barrier throughout the memory hierarchy. (mbar uses the same opcode as eiemo, which is not defined by Book E.) On the e500, msync waits for proceeding data memory accesses to become visible to the entire memory hierarchy; then it is broadcast on the bus. msync completes only after its address tenure is performed without being ARTRYed. Subsequent instructions can execute out of order but complete only after the msync completes. msync latency depends on the processor state when it is dispatched and on various system-level conditions. Frequent use of msync degrades performance. System designs with an external cache should take care to recognize the hardware signaling caused by an MSYNC bus operation and perform the appropriate actions to guarantee that memory references that can be queued internally to the external cache have been performed globally. Note the following: • msync is used to ensure that all stores into a data structure caused by store instructions executed in a critical section of a program are performed with respect to another processor before the store that releases the lock is performed with respect to that processor. mbar is preferable in many cases. • On Motorola Book E devices: Unlike a context synchronizing operation, msync does not discard prefetched instructions.

3.4.1.8 Atomic Update Primitives Using **lwarx** and **stwcx**.

The **lwarx** and **stwcx**. instructions together permit atomic update of a memory location. Book E provides word and double word forms of each of these instructions. Described here is the operation of **lwarx** and **stwcx**.

The specified memory location must require memory coherence if the location may be modified by other processors or mechanisms. If the specified location is in memory that is write-through required or caching inhibited, it is implementation-dependent whether these instructions function correctly or cause the system data storage error handler to be invoked.

The e500 takes a data storage interrupt if the location is write-through but does not take the interrupt if the location is cache-inhibited.

Note the following:

- The memory coherence required attribute on other processors and mechanisms ensures that their stores to the specified location will cause the reservation created by the **lwarx** to be cancelled.
- **Warning:** Support for load and reserve and store conditional instructions for which the specified location is in caching-inhibited memory is being phased out of Book E. It is likely not to be provided on future implementations. New programs should not use these instructions to access caching inhibited memory.

A **lwarx** instruction is a load from a word-aligned location with the following side effects.

- A reservation for a subsequent **stwcx.** instruction is created.
- The memory coherence mechanism is notified that a reservation exists for the location accessed by the **lwarx**.

The **stwcx.** is a store to a word-aligned location that is conditioned on the existence of the reservation created by the **lwarx** and on whether both instructions specify the same location. To emulate an atomic operation, both **lwarx** and **stwcx.** must access the same location. **lwarx** and **stwcx.** are ordered by a dependence on the reservation, and the program is not required to insert other instructions to maintain the order of memory accesses caused by these two instructions.

A **stwcx.** performs a store to the target location only if the location accessed by the **lwarx** that established the reservation has not been stored into by another processor or mechanism between supplying a value for the **lwarx** and storing the value supplied by the **stwcx.** If the instructions specify different locations, the store is not necessarily performed. CR0 is modified to indicate whether the store was performed, as follows:

CR0[LT,GT,EQ,SO] = 0b00 || store_performed || XER[SO]

If a **stwcx.** completes but does not perform the store because a reservation no longer exists, CR0 is modified to indicate that the **stwcx.** completed without altering memory.

A **stwcx.** that performs its store is said to succeed.

Examples using **lwarx** and **stwcx.** are given in Appendix B, “Programming Examples.”

A successful **stwcx.** to a given location may complete before its store has been performed with respect to other processors and mechanisms. As a result, a subsequent load or **lwarx** from the given location on another processor may return a stale value. However, a subsequent **lwarx** from the given location on the other processor followed by a successful **stwcx.** on that processor is guaranteed to have returned the value stored by the first processor’s **stwcx.** (in the absence of other stores to the given location).

3.4.1.8.1 Reservations

The ability to emulate an atomic operation using **lwarx** and **stwcx**. is based on the conditional behavior of **stwcx**., the reservation set by **lwarx**, and the clearing of that reservation if the target location is modified by another processor or mechanism before the **stwcx**. performs its store.

A reservation is held on an aligned unit of real memory called a reservation granule. The size of the reservation granule is implementation-dependent, but is a multiple of 4 bytes for **lwarx**. The reservation granule associated with effective address EA contains the real address to which EA maps. ('real_addr(EA)' in the RTL for the load and reserve and store conditional instructions stands for 'real address to which EA maps.')

When one processor holds a reservation and another processor performs a store, the first processor's reservation is cleared if the store affects any bytes in the reservation granule.

NOTE

One use of **lwarx** and **stwcx**. is to emulate a compare and swap primitive like that provided by the IBM System/370 compare and swap instruction, which See Appendix B, "Programming Examples." checks only that the old and current values of the word being tested are equal, with the result that programs that use such a compare and swap to control a shared resource can err if the word has been modified and the old value is subsequently restored. The use of **lwarx** and **stwcx**. improves on such a compare and swap, because the reservation reliably binds **lwarx** and **stwcx**. together. The reservation is always lost if the word is modified by another processor or mechanism between the **lwarx** and **stwcx**., so the **stwcx**. never succeeds unless the word has not been stored into (by another processor or mechanism) since the **lwarx**.

A processor has at most one reservation at any time. Book E states that a reservation is established by executing a **lwarx** and is lost (or may be lost, in the case of the fourth and fifth bullets) if any of the following occurs.

- The processor holding the reservation executes another **lwarx**: this clears the first reservation and establishes a new one.
- The processor holding the reservation executes any **stwcx**., regardless of whether the specified address matches that of the **lwarx**.
- Some other processor executes a store or **dcbz** to the same reservation granule.
- Some other processor executes a **dcbtst**, **dcbst**, or **dcbf** to the same reservation granule: whether the reservation is lost is undefined.

- Another processor executes a **dcba** to the reservation granule: the reservation is lost if the instruction causes the target block to be newly established in the data cache or to be modified; otherwise whether the reservation is lost is undefined.
- Some other mechanism modifies a location in the same reservation granule.

On the e500, a reservation is lost for any of the following reasons:

- Execution of a **stwcx**.
- Any of the following interrupts occur:
 - External
 - Performance monitor
 - Illegal instruction
 - Critical input interrupt
 - Machine check
 - Fixed interval timer
 - Decrementer
 - Unconditional debug event
 - Watchdog timer
- Snoops
 - Four-state: RWITM, RCLAIM
 - Three- and four-state: writes, flush, kill, dkill)
- Another processor executes any of the following instructions to the reservation granule:
 - **dcbtst** (if four-state cache coherency protocol is used)
 - **dcbf**
 - **dcba**
 - **dcbst** (The e500 broadcasts **dcbst** as a flush; if another processor implements **dcbst** as a clean, the reservation is not cleared)

Interrupts are not guaranteed to clear reservations (however, system software invoked by interrupts may clear reservations).

In general, programming conventions must ensure that **lwarx** and **stwcx**. specify addresses that match; a **stwcx**. should be paired with a specific **lwarx** to the same location. Situations in which a **stwcx**. may erroneously be issued after some **lwarx** other than that with which it is intended to be paired must be scrupulously avoided. For example, there must not be a context switch in which the processor holds a reservation in behalf of the old context, and the new context resumes after a **lwarx** and before the paired **stwcx**.. The **stwcx**. in the new context might succeed, which is not what was intended by the programmer.

Such a situation must be prevented by issuing a **stwcx.** to a dummy writable word-aligned location as part of the context switch, thereby clearing any reservation established by the old context. Executing **stwcx.** to a word-aligned location is enough to clear the reservation, regardless of whether it was set by **lwarx.**

3.4.1.8.2 Forward Progress

Forward progress in loops that use **lwarx** and **stwcx.** is achieved by a cooperative effort among hardware, operating system software, and application software.

Book E guarantees that when a processor executes a **lwarx** to obtain a reservation for location X and then a **stwcx.** to store a value to location X, one of the following:

1. The **stwcx.** succeeds and the value is written to location X
2. The **stwcx.** fails because some other processor or mechanism modified location X
3. The **stwcx.** fails because the processor's reservation was lost for some other reason

In Cases 1 and 2, the system as a whole makes progress in the sense that some processor successfully modifies location X. Case 3 covers reservation loss required for correct operation of the rest of the system. This includes cancellation caused by some other processor writing elsewhere in the reservation granule for X, as well as cancellation caused by the operating system in managing certain limited resources such as real memory or context switches. It may also include implementation-dependent causes of reservation loss.

An implementation may make a forward progress guarantee, defining the conditions under which the system as a whole makes progress. Such a guarantee must specify the possible causes of reservation loss in Case 3. Although Book E alone cannot provide such a guarantee, the conditions in Cases 1 and 2 are necessary for a guarantee. An implementation and operating system can build on them to provide such a guarantee.

Note that Book E does not guarantee fairness. In competing for a reservation, two processors can indefinitely lock out a third.

3.4.1.8.3 Reservation Loss Due to Granularity

Lock words should be allocated such that contention for the locks and updates to nearby data structures do not cause excessive reservation losses due to false indications of sharing that can occur due to the reservation granularity.

A processor holding a reservation on any word in a reservation granule loses its reservation if some other processor stores anywhere in that granule. Such problems can be avoided only by ensuring that few such stores occur. This can most easily be accomplished by allocating an entire granule for a lock and wasting all but one word.

Reservation granularity may vary for each implementation. There are no architectural restrictions bounding the granularity implementations must support, so reasonably portable

code must dynamically allocate aligned and padded memory for locks to guarantee absence of granularity-induced reservation loss.

3.4.1.9 Memory Control Instructions

Memory control instructions can be classified as follows:

- User- and supervisor-level cache management instructions.
- Supervisor-level-only translation lookaside buffer management instructions

This section describes the user-level cache management instructions. See Section 3.4.2.2, “Supervisor-Level Memory Control Instructions,” for information about supervisor-level cache and translation lookaside buffer management instructions.

This section does not describe the cache-locking APU instructions, which are described in Section 3.11.5, “Cache Locking APU.”

3.4.1.9.1 User-Level Cache Instructions

The instructions summarized in this section help user-level programs manage on-chip caches if they are implemented. See Chapter 12, “L1 Caches,” for more information about cache topics. The following sections describe how these operations are treated with respect to the e500’s caches.

As with other memory-related instructions, the effects of cache management instructions on memory are weakly-ordered. If the programmer must ensure that cache or other instructions have been performed with respect to all other processors and system mechanisms, a **msync** instruction must be placed after those instructions.

Note that the e500 interprets cache control instructions (**icbi**, **dcbi**, **dcbf**, **dcbz**, and **dcbst**) as if they pertain only to local caches. A **dcbz** (with M set) is always broadcast on the bus interface if it does not hit as modified in an on-chip cache.

This section does not describe the cache-locking APU instructions, which are described in Section 3.11.5, “Cache Locking APU.”

Note that in those instructions that have a cache target operand (CT); the e500 supports the following values:

- CT = 0 indicates the L1 cache
- CT = 1 indicates the L2 cache

These values are defined by the Motorola Book E Implementation Standards.

Table 3-27 summarizes the user-level cache instructions.

Table 3-27. User-Level Cache Instructions

Name	Mnemonic	Syntax	Implementation Notes
Data Cache Block Touch ¹	dcbt	CT,rA,rB	dcbt allows potential performance enhancements through software-initiated prefetch hints. Implementations are not required to take action based on execution of dcbt but can prefetch the cache block corresponding to the EA into their cache. When dcbt executes, the e500 checks for protection violations (as for a load instruction). dcbt is treated as a no-op in the following cases: • The access causes a protection violation. The page is mapped cache-inhibited. • All lines that this entry maps to are locked or the cache is disabled. • HID0[NOPTI] = 1 Otherwise, if no data is in the cache location, the e500 requests a cache line fill. Data brought into the cache is validated as if it were a load instruction. The memory reference of a dcbt sets the reference bit.
Data Cache Block Touch for Store ¹	dcbtst	CT,rA,rB	dcbtst can be no-oped by setting HID0[NOPTI]. dcbtst behaves similarly to dcbt, except that the line-fill request on the bus is signaled as read or read-claim, and the data is marked as exclusive in the L1 data cache if there is no shared response on the bus. More specifically, the following cases occur depending on where the block currently exists or does not exist in the e500. • dcbtst hits in the L1 data cache. In this case, the dcbtst does nothing and the state of the block in the cache is not changed. Thus, if the block was in the shared state, a subsequent store hits on this shared block and incur the associated latency penalties. • dcbtst misses in the L1 data cache and hits in the L2 cache. In this case, dcbtst reloads the L1 data cache with the state found in the L2 cache. Again, if the block was in the shared state in the L2, a subsequent store hits on this shared block and incur the associated latency penalties. • dcbtst misses in L1 data cache, L2 caches. In this case, e500 requests the block from memory with read or read-claim and reload the L1 data cache in the exclusive state. As subsequent store hits on exclusive and can perform the store to the L1 data cache immediately. dcbtst is no-oped if its target address is mapped as write-through.
Data Cache Block Set to Zero	dcbz	rA,rB	The EA is computed, translated, and checked for protection violations. For cache hits, 32 bytes of zeros are written to the cache block and the tag is marked modified. For cache misses with the replacement block marked not modified, the zero reload is performed and the cache block is marked modified. However, if the replacement block is marked modified, the contents are written back to memory first. dcbz takes an alignment interrupt if the cache is locked or disabled or if the cache is marked WT or CI. If WIMG = xx1x (coherency enforced), the address is broadcast to the bus before the zero reload fill. The interrupt priorities (from highest to lowest) are as follows: 1 Cache disabled—Alignment interrupt 2 Cache is locked—Alignment interrupt 3 Page marked write-through or cache-inhibited—alignment interrupt 4 TLB protection violation—data storage interrupt dcbz is broadcast if WIMG = xx1x (coherency enforced).

Table 3-27. User-Level Cache Instructions (continued)

Name	Mnemonic	Syntax	Implementation Notes
Data Cache Block Allocate	dcba	rA,rB	The EA is computed, translated, and checked for protection violations. For cache hits, 32 bytes of zeros are written to the cache block and the tag is marked modified. For cache misses with the replacement block marked non-dirty, the zero reload is performed and the cache block is marked modified. However, if the replacement block is marked modified, the contents are written back to memory first. dcba performs a no-op if the cache is locked or disabled or if the cache is marked WT or CI. If WIMG = xx1x (coherency enforced), the address is broadcast to the bus before the zero reload fill. A no-op occurs for the following: • Cache is disabled • Cache is locked • Page marked write-through or cache-inhibited • TLB protection violation dcba is broadcast if WIMG = xx1x (coherency enforced).
Data Cache Block Store	dcbst	rA,rB	The EA is computed, translated, and checked for protection violations. • For cache hits with the tag marked not modified, no further action is taken. • For cache hits with the tag marked modified, the cache block is written back to memory and marked exclusive. If WIMG = xx1x (coherency enforced) dcbst is broadcast. dcbst acts like a load with respect to address translation and memory protection. It executes regardless of whether the cache is disabled or locked.
Data Cache Block Flush	dcbf	rA,rB	The EA is computed, translated, and checked for protection violations: • For cache hits with the tag marked modified, the cache block is written back to memory and the cache entry is invalidated. • For cache hits with the tag marked not modified, the entry is invalidated. • For cache misses, no further action is taken. A dcbf is broadcast if WIMG = xx1x (coherency enforced). dcbf acts like a load with respect to address translation and memory protection. It executes regardless of whether the cache is disabled or locked.
Instruction Cache Block Invalidate	icbi	rA,rB	icbi is broadcast on the bus. It should always be followed by an msync and an isync to make sure its effects are seen by instruction fetches following the icbi itself.
Instruction Cache Block Touch	icbt	CT,rA,rB	If CT = 0, the e500 treats icbt as a no-op. If CT = 1, icbt executes as follows: • For L1 data cache hit-to-modified, icbt performs like load on bus, e500 ignores data (for L2) • For L1 data cache hit-to-modified—cast out (for L2)

¹ A program that uses **dcbt** and **dcbtst** improperly is less efficient. To improve performance, HID0[NOPTI] can be set, which causes **dcbt** and **dcbtst** to be no-oped at the cache. They do not cause bus activity and cause only a 1-clock execution latency. The default state of this bit is zero, which enables the use of these instructions.

3.4.2 Supervisor-Level Instructions

The Book E architecture includes the structure of the memory management model, supervisor-level registers, and the interrupt model. This section describes the supervisor-level instructions implemented by e500.

3.4.2.1 System Linkage Instructions

This section describes the system linkage instructions (see Table 3-28). The user-level **sc** instruction lets a user program call on the system to perform a service and causes the processor to take a system call interrupt. The supervisor-level **rfi** instruction is used for returning from an interrupt handler. The **rfci** instruction is used for critical interrupts, and **rfmci** is used for machine check interrupts.

Table 3-28. System Linkage Instructions—Supervisor-Level

Name	Mnemonic	Syntax	Implementation Notes
System Call	sc	—	The sc instruction is context-synchronizing.
Return from Interrupt	rfi	—	rfi is context-synchronizing, which for the e500 means it works its way to the final execute stage, updates architected registers, and redirects the instruction flow.
Return from Critical Interrupt	rfci	—	When rfci is executed, the values in the critical interrupt save and restore registers (CSRR0 and CSRR1) are restored. rfci is context-synchronizing, which for the e500 means it works its way to the final execute stage, updates architected registers, and redirects the instruction flow.
Return from Machine Check Interrupt	rfmci	—	(e500-specific) When rfmci is executed, the values in the machine check interrupt save and restore registers (MCSRR0 and MCSRR1) are restored. rfmci is context-synchronizing; it works its way to the final execute stage, updates architected registers, and redirects instruction flow.

The processor control instructions used to access the MSR and the SPRs are discussed in this section. Table 3-29 lists instructions for accessing the MSR.

Table 3-29. Move to/from Machine State Register Instructions

Name	Mnemonic	Syntax	Description
Move to Machine State Register	mtmsr	rS	—
Move from Machine State Register	mfmsr	rD	—
Write MSR External Enable	wrtee	rS	Bit 48 of the contents of rS is placed into MSR[EE]. No other MSR bits are affected.
Write MSR External Enable Immediate	wrteei	E	The value specified in the E field is placed into MSR[EE]. No other MSR bits are affected.

Certain encodings of the SPR field of **mtspr** and **mfspir** instructions (shown in Table 3-23) provide access to supervisor-level SPRs. Encodings for the architecture-defined SPRs are listed in Table 3-24. Encodings for e500-specific, supervisor-level SPRs are listed in Table 3-25. Simplified mnemonics are provided for **mtspr** and **mfspir**. For a discussion of context synchronization requirements when altering certain SPRs, refer to Appendix B, “Programming Examples.”

3.4.2.2 Supervisor-Level Memory Control Instructions

Memory control instructions include the following:

- Cache management instructions (supervisor-level and user-level)
- Translation lookaside buffer management instructions

This section describes supervisor-level memory control instructions. Section 3.4.1.9, “Memory Control Instructions,” describes user-level memory control instructions.

3.4.2.2.1 Supervisor-Level Cache Instruction

Table 3-30 lists the only supervisor-level cache management instruction.

Table 3-30. Supervisor-Level Cache Management Instruction

Name	Mnemonic	Syntax	Implementation Notes
DataCache Block Invalidate	dcbi	rA,rB	dcbi is executed as described in Book E. The e500 core invalidates the cache block without pushing it out to memory. See Section 3.4.1.9.1, “User-Level Cache Instructions.” On the e500, dcbi cannot generate a cache-locking exception.

See Section 3.4.1.9.1, “User-Level Cache Instructions,” for cache instructions that provide user-level programs the ability to manage the on-chip caches.

3.4.2.2.2 Supervisor-Level TLB Management Instructions

The address translation mechanism is defined in terms of TLBs and page table entries (PTEs) Book E processors use to locate the logical-to-physical address mapping for a particular access.

See Chapter 7, “Memory Management,” for more information about TLB operations. Table 3-31 summarizes the operation of the TLB instructions on the e500 core complex.

Table 3-31. Translation Lookaside Buffer Management Instructions

Name	Mnemonic	Syntax	Implementation Notes
TLB Invalidate Virtual Address Indexed	tlbivax	rA, rB	A TLB invalidate operation is performed whenever tlbivax is executed. tlbivax invalidates any TLB entry that corresponds to the virtual address calculated by this instruction as long as IPROT is not set; this includes invalidating TLB entries contained in TLBs on other processors and devices in addition to the processor executing tlbivax . Thus an invalidate operation is broadcast throughout the coherent domain of the processor executing tlbivax . For more information see Section 13.3, “Translation Lookaside Buffers (TLBs).”
TLB Read Entry	tlbre	—	tlbre causes the contents of a single TLB entry to be extracted from the MMU and be placed in the corresponding fields of the MMU assist (MAS) registers. The entry extracted is specified by the TLBSEL, ESEL and EPN fields of MAS0 and MAS2. The contents extracted from the MMU are placed in MAS0, MAS1, MAS2, and MAS3. See Section 13.3, “Translation Lookaside Buffers (TLBs).” The RTL for the Motorola implementation of tlbre is as follows: <pre> tlb_entry_id = MAS0(TLBSEL, ESEL MAS2 (EPN) result = MMU(tlb_entry_id) MAS0, MAS1, MAS2, MAS3 = result </pre>

Table 3-31. Translation Lookaside Buffer Management Instructions (continued)

Name	Mnemonic	Syntax	Implementation Notes
TLB Search Indexed	tlbsx	rA, rB	tlbsx updates the MAS registers conditionally based on the success or failure of a lookup in the MMU. The lookup is controlled by the EA provided by GPR[rB] specified in the instruction encoding and MAS6[SAS,SPID]. The values placed into MAS0, MAS1, MAS2, and MAS3 differ, depending on whether a successful or unsuccessful search occurred. See Section 13.3, “Translation Lookaside Buffers (TLBs).” The RTL for the e500 implementation of tlbsx is as follows: <pre> if RA!=0 then generate exception EA = ³²0 GPR(RB)_{32:63} ProcessID = MAS6(SPID), 0b0000_0000 AS = MAS6(SAS) VA = AS ProcessID EA if Valid_TLB_matching_entry_exists (VA) then result = see Table 13-14, column “tlbsx hit” else result = see Table 13-14, column “tlbsx miss” MAS0, MAS1, MAS2, MAS3 = result </pre> Note that RA=0 is a preferred form for tlbsx and that some Motorola implementations, such as the e500, take an illegal instruction exception program interrupt if RA!=0.
TLB Write Entry	tlbwe	—	tlbwe causes the contents of certain fields of MAS0, MAS1, MAS2, and MAS3 to be written into a single TLB entry in the MMU. The entry written is specified by the TLBSEL, ESEL, and EPN fields of MAS0 and MAS2. See Section 13.3, “Translation Lookaside Buffers (TLBs).” The RTL for the e500 implementation of tlbwe is as follows: <pre> tlb_entry_id = MAS0(TLBSEL, ESEL) MAS2(EPN) MMU(tlb_entry_id) = MAS0, MAS1, MAS2, MAS3 </pre>
TLB Synchronize	tlbsync	—	Causes a TLBSYNC transaction on the core complex bus. This transaction is retried if any processor, including the one that executed the tlbsync, has pending memory accesses issued before any previous tlbivax completed. See Section 13.3, “Translation Lookaside Buffers (TLBs).”

Implementation Note—The presence and exact semantics of the TLB management instructions are implementation-dependent. To minimize compatibility problems, system software should incorporate uses of these instructions into subroutines.

3.4.3 Recommended Simplified Mnemonics

The description of each instruction includes the mnemonic and a formatted list of operands. Book E-compliant assemblers support the mnemonics and operand lists. To simplify assembly language programming, a set of simplified mnemonics and symbols is provided for some of the most frequently used instructions; refer to Appendix A, “Simplified Mnemonics,” for a complete list. Programs written to be portable across the various assemblers for the Book E architecture should not assume the existence of mnemonics not described in this document.

3.5 Book E Instructions with Implementation-Specific Features

Book E defines several instructions in a general way, leaving the details of the execution up to the implementation. These are listed in Table 3-32. This section describes how the e500 core complex implements those instructions. The implementation specific TLB instructions (listed below) are described in more detail in Section 13.4, “TLB Instructions—Implementation.”

Table 3-32. Implementation-Specific Instructions Summary

Name	Mnemonic	Syntax	Category
Move from APID Indirect	mfapidi	—	Not implemented on the e500. Attempting to execute mfapidi causes an illegal instruction exception type program interrupt.
Move from Device Control Register	mfdcr	—	Not implemented on the e500. Attempting to execute mfdcr causes an illegal instruction exception type program interrupt.
Move to Device Control Register	mtdcr	—	Not implemented on the e500. Attempting to execute mtdcr causes an illegal instruction exception type program interrupt.
TLB Invalidate Virtual Address Indexed	tlbivax	rA, rB	These are generally described in Section 3.4.2.2.2, “Supervisor-Level TLB Management Instructions.” They are described in greater detail in Section 13.4, “TLB Instructions—Implementation.”
TLB Read Entry	tlbre	—	
TLB Search Indexed	tlbsx	rA, rB	
TLB Write Entry	tlbwe	—	

A list of user-level instructions defined by both the classic PowerPC architecture and Book E can be found in Section 3.12, “Instruction Listing.”

3.5.1 New e500 Instructions

The core complex implements the new instructions listed in Table 3-33 that extend the Book E instruction set in accordance with Book E. SPE APU instructions are listed in Table 3-36 and Table 3-37.

Table 3-33. e500-Specific Instructions (Not Including SPE APU Instructions)

Name	Mnemonic	Syntax
Branch Buffer Load Entry and Lock Set	bblels	—
Branch Buffer Entry Lock Reset	bbelr	—
Data Cache Block Lock Clear	dcblc	CT, rA, rB
Data Cache Block Touch and Lock Set	dcbtls	CT, rA, rB
Data Cache Block Touch for Store and Lock Set	dcbtstls	CT, rA, rB
Instruction Cache Block Lock Clear	icblc	CT, rA, rB
Instruction Cache Block Touch and Lock Set	icbtls	CT, rA, rB

Table 3-33. e500-Specific Instructions (Not Including SPE APU Instructions)

Name	Mnemonic	Syntax
Integer Select	isel	rD, rA, rB, crb
Move from Performance Monitor Register	mfpmr	rD,PMRN
Move to Performance Monitor Register	mtpmr	PMRN,rS
Return from Machine Check Interrupt	rfmci	—

3.5.2 Context Synchronization

Context synchronization is achieved by post- and presynchronizing instructions. An instruction is presynchronized by completing all instructions before dispatching the presynchronized instruction. Post-synchronizing is implemented by not dispatching any later instructions until the post-synchronized instruction is completely finished.

3.6 Instruction Fetching

In general, instructions can be prefetched from the cache in the event of a cache hit or from memory in the event of a cache miss. Prefetched instructions may not be executed if the instruction stream is redirected between the time the instructions are fetched and when they are scheduled for execution.

3.7 Branch Prediction

The core complex's instruction fetching mechanism uses a 2-bit saturating up-down counter scheme for branch prediction. The branch path is predicted by a branch target buffer (BTB) and subsequently checked to see if the guess was correct. This enables operation beyond a conditional branch without waiting for the branch to be decoded and resolved. The fetch unit predicts the direction of the branch as follows:

- Guess not taken for any branch whose fetch address misses in the BTB or hits in the BTB and is predicted not taken
- Guess taken for any branch that hits in the BTB and is predicted taken.

3.8 Memory Access Alignment Support

The core complex provides hardware support for unaligned memory accesses; however, there is performance degradation. For loads that hit in the cache, the LSU's throughput degrades to one misaligned load every 3 cycles. Similarly, stores can be translated at a rate of one misaligned store per 3 cycles. Additionally, after translation, each misaligned store is treated as two distinct entries in the store queue, each requiring a cache access.

A word or half-word memory access requires multiple accesses if it crosses a double-word boundary but not if it crosses a natural boundary. SPE vector loads and stores cause alignment interrupts if they cross their natural alignment boundaries (as shown in Table 3-34).

Table 3-34. Natural Alignment Boundaries for Extended Vector Instructions

Instruction	Boundary
evld{d,w,h} evld{d,w,h}x evstd{d,w,h} evstd{d,w,h}x	Double word
evlwwsplat{x} evlwhe{x} evlwhou{x} evlwhos{x} evlwhsplat{x} evstwwe{x} evstwwow{x} evstwhe{x} evstwho{x}	Word
evlhhsplat{x} evlhhouplat{x} evlhhosplat{x}	Half word

Frequent use of unaligned memory accesses is discouraged because of the impact on performance.

Any load word or load half word that crosses a double-word boundary is interruptible, and therefore can restart. If the first access has been performed when the interrupt occurs, it is performed again when the instruction is restarted, even if it is to a page marked as guarded. Any load word or load half that crosses a translation boundary may take a translation exception on the second access. In this case, the first access may have already occurred.

Accesses that cross a translation boundary where the endianness changes cause a byte-ordering data storage interrupt.

3.9 Update Instructions

The load-with-update and store-with-update instructions are described in Book E. Internally, the core complex breaks these instructions into two sub-instructions. The update portion of the instruction is executed by one of the simple ALUs, and the load portion is executed by the load/store unit. Programmers should be aware that the simple ALU used is busy for one cycle executing the update portion of the update instruction.

3.10 Memory Synchronization

The **msync** instruction provides a memory barrier throughout the memory hierarchy. This instruction waits for any preceding data memory accesses to reach the point of coherency (that is, visible to the entire memory hierarchy); then it is broadcast on the core complex bus. Only after the address tenure of the **msync** has been successfully performed on the bus (that is, without being ARTRYed) is **msync** completed. No subsequent instructions in the stream are initiated until after **msync** completes. Note that **msync** uses the same opcode as the **sync** instruction. Syntax for the **msync** instruction is described in Section 3.4.1.7, “Memory Synchronization Instructions.”

The core complex implements two variations of the **mbar** instruction. The desired behavior is selected with the MO field (bits 6–10) of **mbar**, as follows:

- When MO = 0, **mbar** behaves identically to **msync**.
- When MO = 1, **mbar** is a weaker, faster memory barrier that ensures that all data memory accesses caused by instructions preceding it complete before data accesses caused by subsequent instructions. All mechanisms see this order. However, unlike **msync** and **mbar** with MO = 0, **mbar** with MO = 1 does not wait for its address tenure to be performed successfully on the bus before allowing subsequent instructions to complete. This provides a faster way to order data accesses.
- If MO is not 1, the core complex executes **mbar** as though MO = 0.

The core complex implements **lwarx** and **stwcx**, as described in Book E. If the EA is not a multiple of 4 for either instruction, an alignment interrupt is invoked. If either instruction tries to access a page marked as write-through required, a DSI interrupt is invoked.

As specified in Book E, the core complex requires that, for **stwcx**, to succeed, the EA of **stwcx** must be to the same reservation granule as the EA of a preceding **lwarx**. Reservation granularity is implementation-dependent. The core complex makes reservations on behalf of aligned 32-byte sections of the memory address space.

For the purposes of memory coherency, the reservation granule for **lwarx** and **stwcx** is also a cache block. A reservation-killing snoop to any address within a cache block that contains the reservation causes the reservation to be invalidated.

The reservation is invalidated when an external interrupt is signaled.

3.11 e500-Specific Instructions

Instructions that are specific to the e500 core are implemented as auxiliary processing units (APU), and are described in the following sections.

3.11.1 Signal Processing Engine (SPE) APU Instructions

The e500 core complex provides a general-purpose register file with 32, 64-bit registers. The 32-bit Book E instructions operate on the lower (least significant) 32 bits of the 64-bit register. Instructions issue from a single instruction stream. SPE APU vector instructions however treat the 64-bit registers as containing two 32-bit elements or four 16-bit elements as described in Section 3.11.1.1, “SPE APU Vector Instructions.” However, like 32-bit Book E instructions, SPE APU floating-point instructions use bits 32–63 of the GPRs to hold 32-bit single-precision operands, as described in Section 3.11.1.2, “SPE APU Floating-Point Instructions.”

There is no record form of SPE APU instructions. Vector compare instructions store the result of the comparison into the condition register (CR). The meaning of the CR bits are now overloaded for the vector operations. Vector compare instructions specify a CR field and two source registers and also the type of compare: greater than, less than, or equal. Two bits in the CR field are written with the result of the vector compare, one for each element. The two defined bits could be used either by a vector select instruction or by a UISA branch instruction.

A partially visible accumulator register is architected for the integer/fractional SPE APU multiply accumulate forms of instructions. Its usage is described in Section 10.3.1.1.3, “Accumulator.”

Full descriptions of these instructions can be found in the EREF.

3.11.1.1 SPE APU Vector Instructions

New SPE APU (vector) instructions are defined that view the 64-bit register as being composed of a vector of two 32-bit elements, (some of the instructions also read or write 16-bit elements). These new instructions can also perform scalar operations by ignoring the results of the upper 32-bit half of the register file. Some scalar instructions are defined for SPE that produce a 64-bit scalar result.

Table 3-35 shows how SPE APU vector multiply instruction mnemonics are structured.

Table 3-35. SPE APU Vector Multiply Instruction Mnemonic Structure

Prefix	Multiply Element		Data Type Element		Accumulate Element	
evm	ho	half odd (16x16->32)				
	he	half even (16x16->32)				
	heg	half odd guarded (16x16->32)			a	write to ACC
	hog	half even guarded (16x16->32)			aa	write to ACC & added ACC
	wh	word high (32x32->32)	usi	unsigned saturate integer	an	write to ACC & negate ACC
	wl	word low (32x32->32)	umi	unsigned modulo integer	aaw	write to ACC & ACC in words
	whg	word high guarded (32x32->32)	ssi	signed saturate integer	anw	write to ACC & negate ACC in words
	wlg	word low guarded (32x32->32)	ssf	signed saturate fractional		
	w	word (32x32->64)	smi	signed modulo integer		
			smf	signed modulo fractional		

Table 3-36 lists SPE APU vector instructions.

Table 3-36. SPE APU Vector Instructions

Instruction	Mnemonic	Syntax
Bit Reversed Increment	brinc	rD,rA,rB
Vector Absolute Value	evabsw	rD,rA
Vector Add Immediate Word	evaddiw	rD,rB,UIMM
Vector Add Signed, Modulo, Integer to Accumulator Word	evaddsmiaaw	rD,rA,rB
Vector Add Signed, Saturate, Integer to Accumulator Word	evaddssiaaw	rD,rA
Vector Add Unsigned, Modulo, Integer to Accumulator Word	evaddumiaaw	rD,rA
Vector Add Unsigned, Saturate, Integer to Accumulator Word	evaddusiaaw	rD,rA
Vector Add Word	evaddw	rD,rA,rB
Vector AND	evand	rD,rA,rB
Vector AND with Complement	evandc	rD,rA,rB
Vector Compare Equal	evcmpeq	crfD,rA,rB
Vector Compare Greater Than Signed	evcmpgts	crfD,rA,rB
Vector Compare Greater Than Unsigned	evcmpgtu	crfD,rA,rB
Vector Compare Less Than Signed	evcmplt	crfD,rA,rB
Vector Compare Less Than Unsigned	evcmpltu	crfD,rA,rB
Vector Count Leading Sign Bits Word	evcntlsw	rD,rA
Vector Count Leading Zeros Word	evcntlzw	rD,rA
Vector Divide Word Signed	evdivws	rD,rA,rB
Vector Divide Word Unsigned	evdivwu	rD,rA,rB
Vector Equivalent	eveqv	rD,rA,rB
Vector Extend Sign Byte	evextsb	rD,rA
Vector Extend Sign Half Word	evextsh	rD,rA
Vector Floating-Point Absolute Value	evfsabs	rD,rA
Vector Floating-Point Add	evfsadd	rD,rA,rB
Vector Convert Floating-Point from Signed Fraction	evfscfsf	rD,rB
Vector Convert Floating-Point from Signed Integer	evfscfsi	rD,rB
Vector Convert Floating-Point from Unsigned Fraction	evfscfuf	rD,rB
Vector Convert Floating-Point from Unsigned Integer	evfscfui	rD,rB
Vector Floating-Point Compare Equal	evfscmpcq	crfD,rA,rB
Vector Floating-Point Compare Greater Than	evfscmpgt	crfD,rA,rB
Vector Floating-Point Compare Less Than	evfscmplt	crfD,rA,rB
Vector Convert Floating-Point to Signed Fraction	evfsctsf	rD,rB

Table 3-36. SPE APU Vector Instructions (continued)

Instruction	Mnemonic	Syntax
Vector Convert Floating-Point to Signed Integer	evfsctsi	rD,rB
Vector Convert Floating-Point to Signed Integer with Round toward Zero	evfsctsiz	rD,rB
Vector Convert Floating-Point to Unsigned Fraction	evfsctuf	rD,rB
Vector Convert Floating-Point to Unsigned Integer	evfsctui	rD,rB
Vector Convert Floating-Point to Unsigned Integer with Round toward Zero	evfsctuiz	rD,rB
Vector Floating-Point Divide	evfsdiv	rD,rA,rB
Vector Floating-Point Multiply	evfsmul	rD,rA,rB
Vector Floating-Point Negative Absolute Value	evfsnabs	rD,rA
Vector Floating-Point Negate	evfsneg	rD,rA
Vector Floating-Point Subtract	evfssub	rD,rA,rB
Vector Floating-Point Test Equal	evfststeq	crfD,rA,rB
Vector Floating-Point Test Greater Than	evfststgt	crfD,rA,rB
Vector Floating-Point Test Less Than	evfststlt	crfD,rA,rB
Vector Load Double Word into Double Word	evldd	rD,d(rA)
Vector Load Double Word into Double Word Indexed	evlddx	rD,rA,rB
Vector Load Double into Half Words	evldh	rD,d(rA)
Vector Load Double into Half Words Indexed	evldhx	rD,rA,rB
Vector Load Double into Two Words	evldw	rD,d(rA)
Vector Load Double into Two Words Indexed	evldwx	rD,rA,rB
Vector Load Half Word into Half Words Even and Splat	evlhhesplat	rD,d(rA)
Vector Load Half Word into Half Words Even and Splat Indexed	evlhhesplatx	rD,rA,rB
Vector Load Half Word into Half Word Odd Signed and Splat	evlhhosspat	rD,d(rA)
Vector Load Half Word into Half Word Odd Signed and Splat Indexed	evlhhosspatx	rD,rA,rB
Vector Load Half Word into Half Word Odd Unsigned and Splat	evlhhoussplat	rD,d(rA)
Vector Load Half Word into Half Word Odd Unsigned and Splat Indexed	evlhhoussplatx	rD,rA,rB
Vector Load Word into Two Half Words Even	evlwhe	rD,d(rA)
Vector Load Word into Two Half Words Even Indexed	evlwhex	rD,rA,rB
Vector Load Word into Half Words Odd Signed (with sign extension)	evlw hos	rD,d(rA)
Vector Load Word into Half Words Odd Signed Indexed (with sign extension)	evlw hosx	rD,rA,rB
Vector Load Word into Two Half Words Odd Unsigned (zero-extended)	evlw hou	rD,d(rA)
Vector Load Word into Two Half Words Odd Unsigned Indexed (zero-extended)	evlw hous	rD,rA,rB
Vector Load Word into Half Words and Splat	evlw hsplat	rD,d(rA)
Vector Load Word into Half Words and Splat Indexed	evlw hsplatx	rD,rA,rB

Table 3-36. SPE APU Vector Instructions (continued)

Instruction	Mnemonic	Syntax
Vector Load Word into Word and Splat	evlwwsplat	rD,d(rA)
Vector Load Word into Word and Splat Indexed	evlwwsplatx	rD,rA,rB
Vector Merge High	evmergehi	rD,rA,rB
Vector Merge High/Low	evmergehilo	rD,rA,rB
Vector Merge Low	evmergelo	rD,rA,rB
Vector Merge Low/High	evmergelohi	rD,rA,rB
Multiply Half Words, Even, Guarded, Signed, Modulo, Fractional and Accumulate	evmhegsmfaa	rD,rA,rB
Multiply Half Words, Even, Guarded, Signed, Modulo, Fractional and Accumulate Negative	evmhegsmfan	rD,rA,rB
Multiply Half Words, Even, Guarded, Signed, Modulo, Integer and Accumulate	evmhegsmiaa	rD,rA,rB
Multiply Half Words, Even, Guarded, Signed, Modulo, Integer and Accumulate Negative	evmhegsmian	rD,rA,rB
Multiply Half Words, Even, Guarded, Unsigned, Modulo, Integer and Accumulate	evmhegumiaa	rD,rA,rB
Multiply Half Words, Even, Guarded, Unsigned, Modulo, Integer and Accumulate Negative	evmhegumian	rD,rA,rB
Vector Multiply Half Words, Even, Signed, Modulo, Fractional	evmhesmf	rD,rA,rB
Vector Multiply Half Words, Even, Signed, Modulo, Fractional, Accumulate	evmhesmfa	rD,rA,rB
Vector Multiply Half Words, Even, Signed, Modulo, Fractional and Accumulate into Words	evmhesmfaaw	rD,rA,rB
Vector Multiply Half Words, Even, Signed, Modulo, Fractional and Accumulate Negative into Words	evmhesmfanw	rD,rA,rB
Vector Multiply Half Words, Even, Signed, Modulo, Integer	evmhesmi	rD,rA,rB
Vector Multiply Half Words, Even, Signed, Modulo, Integer, Accumulate	evmhesmia	rD,rA,rB
Vector Multiply Half Words, Even, Signed, Modulo, Integer and Accumulate into Words	evmhesmiaaw	rD,rA,rB
Vector Multiply Half Words, Even, Signed, Modulo, Integer and Accumulate Negative into Words	evmhesmianw	rD,rA,rB
Vector Multiply Half Words, Even, Signed, Saturate, Fractional	evmhessf	rD,rA,rB
Vector Multiply Half Words, Even, Signed, Saturate, Fractional, Accumulate	evmhessfa	rD,rA,rB
Vector Multiply Half Words, Even, Signed, Saturate, Fractional and Accumulate into Words	evmhessfaaw	rD,rA,rB
Vector Multiply Half Words, Even, Signed, Saturate, Fractional and Accumulate Negative into Words	evmhessfanw	rD,rA,rB
Vector Multiply Half Words, Even, Signed, Saturate, Integer and Accumulate into Words	evmhessiaaw	rD,rA,rB
Vector Multiply Half Words, Even, Signed, Saturate, Integer and Accumulate Negative into Words	evmhessianw	rD,rA,rB
Vector Multiply Half Words, Even, Unsigned, Modulo, Integer	evmheumi	rD,rA,rB

Table 3-36. SPE APU Vector Instructions (continued)

Instruction	Mnemonic	Syntax
Vector Multiply Half Words, Even, Unsigned, Modulo, Integer, Accumulate	evmheumia	rD,rA,rB
Vector Multiply Half Words, Even, Unsigned, Modulo, Integer and Accumulate into Words	evmheumiaaw	rD,rA,rB
Vector Multiply Half Words, Even, Unsigned, Modulo, Integer and Accumulate Negative into Words	evmheumianw	rD,rA,rB
Vector Multiply Half Words, Even, Unsigned, Saturate, Integer and Accumulate into Words	evmheusiaaw	rD,rA,rB
Vector Multiply Half Words, Even, Unsigned, Saturate, Integer and Accumulate Negative into Words	evmheusianw	rD,rA,rB
Multiply Half Words, Odd, Guarded, Signed, Modulo, Fractional and Accumulate	evmhogsmfaa	rD,rA,rB
Multiply Half Words, Odd, Guarded, Signed, Modulo, Fractional and Accumulate Negative	evmhogsmfan	rD,rA,rB
Multiply Half Words, Odd, Guarded, Signed, Modulo, Integer and Accumulate	evmhogsmiaa	rD,rA,rB
Multiply Half Words, Odd, Guarded, Signed, Modulo, Integer and Accumulate Negative	evmhogsmian	rD,rA,rB
Multiply Half Words, Odd, Guarded, Unsigned, Modulo, Integer and Accumulate	evmhogumiaa	rD,rA,rB
Multiply Half Words, Odd, Guarded, Unsigned, Modulo, Integer and Accumulate Negative	evmhogumian	rD,rA,rB
Vector Multiply Half Words, Odd, Signed, Modulo, Fractional	evmhosmf	rD,rA,rB
Vector Multiply Half Words, Odd, Signed, Modulo, Fractional, Accumulate	evmhosmfa	rD,rA,rB
Vector Multiply Half Words, Odd, Signed, Modulo, Fractional and Accumulate into Words	evmhosmfaaw	rD,rA,rB
Vector Multiply Half Words, Odd, Signed, Modulo, Fractional and Accumulate Negative into Words	evmhosmfanw	rD,rA,rB
Vector Multiply Half Words, Odd, Signed, Modulo, Integer	evmhosmi	rD,rA,rB
Vector Multiply Half Words, Odd, Signed, Modulo, Integer, Accumulate	evmhosmia	rD,rA,rB
Vector Multiply Half Words, Odd, Signed, Modulo, Integer and Accumulate into Words	evmhosmiaaw	rD,rA,rB
Vector Multiply Half Words, Odd, Signed, Modulo, Integer and Accumulate Negative into Words	evmhosmianw	rD,rA,rB
Vector Multiply Half Words, Odd, Signed, Saturate, Fractional	evmhossf	rD,rA,rB
Vector Multiply Half Words, Odd, Signed, Saturate, Fractional, Accumulate	evmhossfa	rD,rA,rB
Vector Multiply Half Words, Odd, Signed, Saturate, Fractional and Accumulate into Words	evmhossfaaw	rD,rA,rB
Vector Multiply Half Words, Odd, Signed, Saturate, Fractional and Accumulate Negative into Words	evmhossfanw	rD,rA,rB
Vector Multiply Half Words, Odd, Signed, Saturate, Integer and Accumulate into Words	evmhossiaaw	rD,rA,rB

Table 3-36. SPE APU Vector Instructions (continued)

Instruction	Mnemonic	Syntax
Vector Multiply Half Words, Odd, Signed, Saturate, Integer and Accumulate Negative into Words	evmhossianw	rD,rA,rB
Vector Multiply Half Words, Odd, Unsigned, Modulo, Integer	evmhoumi	rD,rA,rB
Vector Multiply Half Words, Odd, Unsigned, Modulo, Integer, Accumulate	evmhoumia	rD,rA,rB
Vector Multiply Half Words, Odd, Unsigned, Modulo, Integer and Accumulate into Words	evmhoumiaaw	rD,rA,rB
Vector Multiply Half Words, Odd, Unsigned, Modulo, Integer and Accumulate Negative into Words	evmhoumianw	rD,rA,rB
Vector Multiply Half Words, Odd, Unsigned, Saturate, Integer and Accumulate into Words	evmhousiaaw	rD,rA,rB
Vector Multiply Half Words, Odd, Unsigned, Saturate, Integer and Accumulate Negative into Words	evmhousianw	rD,rA,rB
Initialize Accumulator	evmra	rD,rA
Vector Multiply Word High Signed, Modulo, Fractional	evmwhsmf	rD,rA,rB
Vector Multiply Word High Signed, Modulo, Fractional and Accumulate	evmwhsmfa	rD,rA,rB
Vector Multiply Word High Signed, Modulo, Integer	evmwhsmi	rD,rA,rB
Vector Multiply Word High Signed, Modulo, Integer and Accumulate	evmwhsmia	rD,rA,rB
Vector Multiply Word High Signed, Saturate, Fractional	evmwhssf	rD,rA,rB
Vector Multiply Word High Signed, Saturate, Fractional and Accumulate	evmwhssfa	rD,rA,rB
Vector Multiply Word High Unsigned, Modulo, Integer	evmwhumi	rD,rA,rB
Vector Multiply Word High Unsigned, Modulo, Integer and Accumulate	evmwhumia	rD,rA,rB
Vector Multiply Word Low Signed, Modulo, Fractional	evmwlsmf	rD,rA,rB
Vector Multiply Word Low Signed, Modulo, Fractional and Accumulate	evmwlsmfafa	rD,rA,rB
Vector Multiply Word Low Signed, Modulo, Fractional and Accumulate in Words	evmwlsmfafaaw	rD,rA,rB
Vector Multiply Word Low Signed, Modulo, Fractional and Accumulate Negative in Words	evmwlsmfafanw	rD,rA,rB
Vector Multiply Word Low Signed, Modulo, Integer and Accumulate in Words	evmwlsmiaaw	rD,rA,rB
Vector Multiply Word Low Signed, Modulo, Integer and Accumulate Negative in Words	evmwlsmianw	rD,rA,rB
Vector Multiply Word Low Signed, Saturate, Fractional	evmwlssf	rD,rA,rB
Vector Multiply Word Low Signed, Saturate, Fractional and Accumulate	evmwlssfa	rD,rA,rB
Vector Multiply Word Low Signed, Saturate, Fractional and Accumulate in Words	evmwlssfaaw	rD,rA,rB
Vector Multiply Word Low Signed, Saturate, Fractional and Accumulate Negative in Words	evmwlssfanw	rD,rA,rB
Vector Multiply Word Low Signed, Saturate, Integer and Accumulate in Words	evmwlssiaaw	rD,rA,rB
Vector Multiply Word Low Signed, Saturate, Integer and Accumulate Negative in Words	evmwlssianw	rD,rA,rB

Table 3-36. SPE APU Vector Instructions (continued)

Instruction	Mnemonic	Syntax
Vector Multiply Word Low Unsigned, Modulo, Integer	evmwlsmi	rD,rA,rB
Vector Multiply Word Low Unsigned, Modulo, Integer and Accumulate	evmwlumia	rD,rA,rB
Vector Multiply Word Low Unsigned, Modulo, Integer and Accumulate in Words	evmwlumiaaw	rD,rA,rB
Vector Multiply Word Low Unsigned, Modulo, Integer and Accumulate Negative in Words	evmwlumianw	rD,rA,rB
Vector Multiply Word Low Unsigned, Saturate, Integer and Accumulate in Words	evmwlusiaaw	rD,rA,rB
Vector Multiply Word Low Unsigned, Saturate, Integer and Accumulate Negative in Words	evmwlusianw	rD,rA,rB
Vector Multiply Word Signed, Modulo, Fractional	evmwsmf	rD,rA,rB
Vector Multiply Word Signed, Modulo, Fractional and Accumulate	evmwsmfa	rD,rA,rB
Vector Multiply Word Signed, Modulo, Fractional and Accumulate	evmwsmfaa	rD,rA,rB
Vector Multiply Word Signed, Modulo, Fractional and Accumulate Negative	evmwsmfan	rD,rA,rB
Vector Multiply Word Signed, Modulo, Integer	evmwsmi	rD,rA,rB
Vector Multiply Word Signed, Modulo, Integer and Accumulate	evmwsmia	rD,rA,rB
Vector Multiply Word Signed, Modulo, Integer and Accumulate	evmwsmiaa	rD,rA,rB
Vector Multiply Word Signed, Modulo, Integer and Accumulate Negative	evmwsmian	rD,rA,rB
Vector Multiply Word Signed, Saturate, Fractional	evmwssf	rD,rA,rB
Vector Multiply Word Signed, Saturate, Fractional and Accumulate	evmwssf	rD,rA,rB
Vector Multiply Word Signed, Saturate, Fractional and Accumulate	evmwssfaa	rD,rA,rB
Vector Multiply Word Signed, Saturate, Fractional and Accumulate Negative	evmwssfan	rD,rA,rB
Vector Multiply Word Unsigned, Modulo, Integer	evmwumi	rD,rA,rB
Vector Multiply Word Unsigned, Modulo, Integer and Accumulate	evmwumia	rD,rA,rB
Vector Multiply Word Unsigned, Modulo, Integer and Accumulate	evmwumiaa	rD,rA,rB
Vector Multiply Word Unsigned, Modulo, Integer and Accumulate Negative	evmwumian	rD,rA,rB
Vector NAND	evnand	rD,rA,rB
Vector Negate	evneg	rD,rA
Vector NOR	evnor	rD,rA,rB
Vector OR	evor	rD,rA,rB
Vector OR with Complement	evorc	rD,rA,rB
Vector Rotate Left Word	evrlw	rD,rA,rB
Vector Rotate Left Word Immediate	evrlwi	rD,rA,UIMM
Vector Round Word	evrndw	rD,rA
Vector Select	evsel	rD,rA,rB,crfS
Vector Shift Left Word	evslw	rD,rA,rB

Table 3-36. SPE APU Vector Instructions (continued)

Instruction	Mnemonic	Syntax
Vector Shift Left Word Immediate	evslwi	rD,rA,UIMM
Vector Splat Fractional Immediate	evsplatfi	rD,SIMM
Vector Splat Immediate	evsplati	rD,SIMM
Vector Shift Right Word Immediate Signed	evsrwis	rD,rA,UIMM
Vector Shift Right Word Immediate Unsigned	evsrwiu	rD,rA,UIMM
Vector Shift Right Word Signed	evsrws	rD,rA,rB
Vector Shift Right Word Unsigned	evsrwu	rD,rA,rB
Vector Store Double of Double	evstdd	rS,d(rA)
Vector Store Double of Double Indexed	evstddx	rS,rA,rB
Vector Store Double of Four Half Words	evstdh	rS,d(rA)
Vector Store Double of Four Half Words Indexed	evstdhx	rS,rA,rB
Vector Store Double of Two Words	evstdw	rS,d(rA)
Vector Store Double of Two Words Indexed	evstdwx	rS,rA,rB
Vector Store Word of Two Half Words from Even	evstwhe	rS,d(rA)
Vector Store Word of Two Half Words from Even Indexed	evstwhex	rS,rA,rB
Vector Store Word of Two Half Words from Odd	evstwho	rS,d(rA)
Vector Store Word of Two Half Words from Odd Indexed	evstwhox	rS,rA,rB
Vector Store Word of Word from Even	evstwwex	rS,d(rA)
Vector Store Word of Word from Even Indexed	evstwwex	rS,rA,rB
Vector Store Word of Word from Odd	evstwwo	rS,d(rA)
Vector Store Word of Word from Odd Indexed	evstwwox	rS,rA,rB
Vector Subtract Signed, Modulo, Integer to Accumulator Word	evsubfsmiaaw	rD,rA
Vector Subtract Signed, Saturate, Integer to Accumulator Word	evsubfssiaaw	rD,rA
Vector Subtract Unsigned, Modulo, Integer to Accumulator Word	evsubfumiaaw	rD,rA
Vector Subtract Unsigned, Saturate, Integer to Accumulator Word	evsubfusiaaw	rD,rA
Vector Subtract from Word	evsubfw	rD,rA,rB
Vector Subtract Immediate from Word	evsubifw	rD,UIMM,rB
Vector XOR	evxor	rD,rA,rB

3.11.1.2 SPE APU Floating-Point Instructions

The SPE APU defines instructions that perform floating-point operations on single-precision operands. SPE APU floating-point operations are IEEE-compliant with software exception handlers and offer a simpler exception model than the floating-point instructions defined by the PowerPC ISA. These instructions use GPRs instead of FPRs,

which offer improved performance for converting between floating-point, integer, and fractional values. Sharing the GPRs allows the SPE APU floating-point instructions to use vector load and store operands.

The SPE APU implements two matching sets of instructions, as follows:

- Vector floating-point instructions operate on a vector of two 32-bit, single-precision floating-point numbers that reside in the upper and lower halves of the 64-bit GPRs. These instructions are listed in Table 3-36 as part of the SPE APU vector instruction set. These are also shown in Table 3-37 alongside their scalar equivalents.
- Scalar floating-point instructions operate on single 32-bit operands that reside in the lower 32-bits of the GPRs. These instructions are considered a proper subset of the SPE APU. These instructions are listed in Table 3-37.

Note that both the vector and scalar versions of the instructions have the same syntax.

Table 3-37. SPE APU Scalar Floating-Point Instructions

Instruction	Mnemonic		Syntax
	Scalar	Vector Equivalent	
Floating-Point Absolute Value	efsabs	evfsabs	rD,rA
Floating-Point Add	efsadd	evfsadd	rD,rA,rB
Convert Floating-Point from Signed Fraction	efscfsf	evfscfsf	rD,rB
Convert Floating-Point from Signed Integer	efscfsi	evfscfsi	rD,rB
Convert Floating-Point from Unsigned Fraction	efscfuf	evfscfuf	rD,rB
Convert Floating-Point from Unsigned Integer	efscfui	evfscfui	rD,rB
Floating-Point Compare Equal	efscmpeq	evfscmpeq	crfD,rA,rB
Floating-Point Compare Greater Than	efscmpgt	evfscmpgt	crfD,rA,rB
Floating-Point Compare Less Than	efscmplt	evfscmplt	crfD,rA,rB
Convert Floating-Point to Signed Fraction	efscfsf	evfscfsf	rD,rB
Convert Floating-Point to Signed Integer	efscfsi	evfscfsi	rD,rB
Convert Floating-Point to Signed Integer with Round toward Zero	efscfsiz	evfscfsiz	rD,rB
Convert Floating-Point to Unsigned Fraction	efscfuf	evfscfuf	rD,rB
Convert Floating-Point to Unsigned Integer	efscfui	evfscfui	rD,rB
Convert Floating-Point to Unsigned Integer with Round toward Zero	efscfuiz	evfscfuiz	rD,rB
Floating-Point Divide	efsdiv	evfsdiv	rD,rA,rB
Floating-Point Multiply	efsmul	evfsmul	rD,rA,rB
Floating-Point Negative Absolute Value	efsnabs	evfsnabs	rD,rA
Floating-Point Negate	efsneg	evfsneg	rD,rA
Floating-Point Subtract	efssub	evfssub	rD,rA,rB

Table 3-37. SPE APU Scalar Floating-Point Instructions

Instruction	Mnemonic		Syntax
	Scalar	Vector Equivalent	
Floating-Point Test Equal	efststeq	evfststeq	crfD,rA,rB
Floating-Point Test Greater Than	efststgt	evfststgt	crfD,rA,rB
Floating-Point Test Less Than	efststlt	evfststlt	crfD,rA,rB

3.11.2 Branch Target Buffer (BTB) Locking Instructions

The core complex provides a 512-entry branch target buffer (BTB) for efficient processing of branch instructions. The BTB is a branch target address cache (BTAC), organized as 128 rows with four-way set associativity, that holds the address of the target instruction of the 512 most-recently taken branches. Table 3-38 lists the BTB instructions.

Table 3-38. Branch Target Buffer (BTB) Instructions

Name	Mnemonic	Syntax
Branch Buffer Load Entry and Lock Set	bblels	—
Branch Buffer Entry Lock Reset	bbelr	—

The core complex also provides support for locking and unlocking entries into the BTB for deterministic branch behavior.

The branch buffer entry address register (BBEAR) and the branch buffer target address register (BBTAR) are defined in the core complex for enabling the locking and unlocking of BTB entries. They can be read and written in both user and supervisor modes with **mfspr** and **mtspr**. A new MSR bit, called the user branch locking enable (UBLE) bit, is also defined to allow user mode programs to lock or unlock entries in the BTB. See Chapter 4, “Execution Timing.”

3.11.3 Instruction Select (isel) APU

The instruction select APU consists of a single instruction (**isel**), which is a conditional register move that helps eliminate branches. Use of this APU is described in Section 10.1, “Instruction Select APU.”

Table 3-39. Instruction Select APU Instruction

Name	Mnemonic	Syntax
Instruction Select	isel	rD,rA,rB,crB

3.11.4 Performance Monitor APU

The e500 core complex implements a performance monitor as an APU. Software communication with the performance monitor APU is achieved through performance monitor registers (PMRs) rather than SPRs. New instructions are provided to move to and from these PMRs. Performance monitor APU instructions are described in Table 3-40. Full descriptions of these instructions can be found in the EREF Chapter 3, “Instruction Set.”

Table 3-40. Performance Monitor APU Instructions

Name	Mnemonic	Syntax
Move from Performance Monitor Register	mfpmr	rD,PMRN
Move to Performance Monitor Register	mtpmr	PMRN,rS

PMR encodings are listed in Table 3-41.

Table 3-41. e500-Defined PMR Encodings

Register Name	PMR			Privilege	Access
	Decimal	pmr[5–9]	pmr[0–4]		
UMMCR0	936	11101	01000	User	Read-only
UMMCR1	940	11101	01100	User	Read-only
UMMCR2	928	11101	00111	User	Read-only
UPMC1	937	11101	01001	User	Read-only
UPMC2	938	11101	01010	User	Read-only
UPMC3	941	11101	01101	User	Read-only
UPMC4	942	11101	01110	User	Read-only
USIAR	939	11101	01011	User	Read-only

3.11.5 Cache Locking APU

This section describes the instructions in the cache locking APU, which consists of the instructions described in Table 3-42. Full descriptions of these instructions can be found in the EREF Chapter 3, “Instruction Set.”

Note the following:

- In the L1 data cache the e500 implements a lock bit for every index and way, allowing a line locking granularity. Note that the e500 supports only CT=0 and CT=1. If CT=0, the L1 data cache is the target cache. If CT=1, the unified L2 cache is the target cache. If CT is not 0 or 1, the instruction is treated as a no-op.
- Note that setting L1CSR0[DCLFI] flash invalidates all data cache lock bits and L1CSR0[ICLFI] flash invalidates all instruction cache lock bits, allowing system

software to clear all cache locking in the L1 cache without knowing the addresses of the lines locked.

- Overlocking occurs when **dcbtls**, **dcbtstls**, or **icbtl**s is performed to an index in either the L1 or L2 cache that already has all ways locked. On the e500, overlocking does not generate an exception; instead, if a touch and lock-set is performed with CT = 0 to an index in which all cache ways are already locked, the least recently used way is evicted and L1CSR0[DCLO] is set indicating an overlock; the new line is not locked or cached.

To precisely detect an overlock condition in the data cache, system software must perform the following code sequence:

```
dcbtls
msync
mf spr [L1CSR0]
    (check L1CSR0[DCUL] for data cache index unable-to-lock condition)
    (check L1CSR0[DCLO] for data cache index over-lock condition)
```

The following code sequence is used to precisely detect an overlock in the instruction cache:

```
icbtl
msync
mf spr (L1CSR1)
    check L1CSR1[ICUL] for instruction cache index unable-to-lock condition
    check L1CSR1[ICLO] for instruction cache index over-lock condition
```

- Touch and lock set instructions (**icbtl**s, **dcbtls**, and **dcbtstls**) are always executed and are not treated as hints. When one of these instructions is performed to an index and the way cannot be locked, L1CSR1[ICUL] or L1CSR0[DCUL] is set to indicate an unable-to-lock condition. This occurs if the instruction must be NOPed.

Table 3-42. Cache Locking APU Instructions

Name	Mnemonic	Syntax	Implementation Details
Data Cache Block Lock Clear	dcbcl	CT,rA,rB	If CT=0 and the line is in the L1 data cache, the data cache lock bit for that line is cleared, making it eligible for replacement. If CT=1 and the line is in the L2 cache, the lock bit for that line is cleared, making it eligible for replacement.
Data Cache Block Touch and Lock Set	dcbtls	CT,rA,rB	If CT=0, the line is loaded and locked into the L1 data cache. If CT=1, the line is loaded and locked in the unified L2 cache. If CT=1 and the block is already in the L2 cache, dcbtls marks the block so it is not a candidate for replacement.
Data Cache Block Touch for Store and Lock Set	dcbtstls	CT,rA,rB	If CT = 0, the e500 core fetches the block containing the byte addressed by EA into the data cache. After the block containing the byte is fetched, it is locked. If CT = 0 and the block is in the data cache, dcbtstls marks the block locked so it is no longer eligible for replacement. If CT=1 and the block is in the L2 cache, dcbtstls marks the block such that it should not be selected for replacement. If CT is not 0 or 1, dcbtstls is NOPed. In the L1 data cache, the e500 implements a lock bit for every index and way, allowing a line locking granularity.

Table 3-42. Cache Locking APU Instructions (continued)

Name	Mnemonic	Syntax	Implementation Details
Instruction Cache Block Lock Clear	icblc	CT,rA,rB	If CT=0 and the line is in the instruction cache, the lock bit for that line is cleared, making it eligible for replacement. If CT=1 and the line is in the L2 cache, the lock bit for that line is cleared in the L2 cache, making it eligible for replacement. If CT is not 0 or 1, the icblc is NOPed.
Instruction Cache Block Touch and Lock Set	icbtlc	CT,rA,rB	If CT=0, the line is loaded and locked into the L1 data cache. If CT=1, the line is loaded into the unified L2 cache and the line is locked into the L2 cache. If CT=1 and the block already exists in the L2 cache, icbtlc marks it such that it should not be selected for replacement. If CT is not 0 or 1, icbtlc is NOPed.

The e500 implements a flash clear for all the lock bits in the data cache (using L1CSR0[CLFR]) and in the instruction cache (using L1CSR1[ICLFR]). This allows system software to clear all data cache locking bits without knowing the addresses of the lines locked.

3.11.6 Machine Check APU

The machine check APU defines a separate interrupt type, like the Book E–defined critical interrupt, for machine check interrupts. It provides additional save and restore SPRs (MCSR0 and MCSRR1) as well as the Return from Machine Check Interrupt instruction (**rfmci**), which is described in Table 3-43.

Table 3-43. Machine Check APU Instruction

Name	Mnemonic	Syntax	Implementation Notes
Return from Machine Check Interrupt	rfmci	—	When rfmci is executed, the values in the machine check save and restore registers (MCSR0 and MCSRR1) are restored. rfmci is context-synchronizing; it works its way to the final execute stage, updates architected registers, and redirects instruction flow.

3.12 Instruction Listing

Table 3-44 lists instructions that are defined in Book E, in the PowerPC architecture, and in the e500. An x or text in a column indicates that the instruction is defined or implemented. The e500-specific instructions are indicated in the e500 column by the name of the facility (BTB, SPE APU, SPE floating-point unit (SPFPU), cache locking) that defines the instruction.

Table 3-44. List of Instructions

Mnemonic	Book E	PowerPC AIM	e500
add	x	x	x
add.	x	x	x

Table 3-44. List of Instructions (continued)

Mnemonic	Book E	PowerPC AIM	e500
addc	x	x	x
addc.	x	x	x
addco	x	x	x
addco.	x	x	x
adde	x	x	x
adde.	x	x	x
adde64	64-bit only		
adde64o	64-bit only		
addeo	x	x	x
addeo.	x	x	x
addi	x	x	x
addic	x	x	x
addic.	x	x	x
addis	x	x	x
addme	x	x	x
addme.	x	x	x
addme64	64-bit only		
addme64o	64-bit only		
addmeo	x	x	x
addmeo.	x	x	x
addo	x	x	x
addo.	x	x	x
addze	x	x	x
addze.	x	x	x
addze64	64-bit only		
addze64o	64-bit only		
addzeo	x	x	x
addzeo.	x	x	x
and	x	x	x
and.	x	x	x
andc	x	x	x
andc.	x	x	x
andi	x	x	x

Table 3-44. List of Instructions (continued)

Mnemonic	Book E	PowerPC AIM	e500
andis.	x	x	x
b	x	x	x
ba	x	x	x
bbeir			BTB
bbleis			BTB
bc	x	x	x
bca	x	x	x
bcctr	x	x	x
bcctre	64-bit only		
bcctrel	64-bit only		
bcctrl	x	x	x
bce	64-bit only		
bcea	64-bit only		
bcel	64-bit only		
bcela	64-bit only		
bcl	x	x	x
bcla	x	x	x
bclr	x	x	x
bclre	64-bit only		
bclrel	64-bit only		
bclrl	x	x	x
be	64-bit only		
bea	64-bit only		
bel	64-bit only		
bela	64-bit only		
bl	x	x	x
bla	x	x	x
brinc			SPE APU
cmp	x	x	x
cmpi	x	x	x
cmpl	x	x	x
cmpli	x	x	x
cntlzd	64-bit only	64-bit only	

Table 3-44. List of Instructions (continued)

Mnemonic	Book E	PowerPC AIM	e500
cntlzd.	64-bit only	64-bit only	
cntlzw	x	x	x
cntlzw.	x	x	x
crand	x	x	x
crandc	x	x	x
creqv	x	x	x
crnand	x	x	x
crnor	x	x	x
cror	x	x	x
crorc	x	x	x
crxor	x	x	x
dcba	x	x	x
dcbae	64-bit only		
dcbf	x	x	x
dcbfе	64-bit only		
dcbi	x	x	x
dcbie	64-bit only		
dcblc			Cache locking
dcblce			Cache locking (64-bit only)
dcbst	x	x	x
dcbstе	64-bit only		
dcbt	x	x	x
dcbte	64-bit only		
dcbtls			Cache locking
dcbtlse			Cache locking (64-bit only)
dcbtst	x	x	x
dcbtste	64-bit only		
dcbtstls			Cache locking
dcbtstlse			Cache locking (64-bit only)
dcbz	x	x	x
dcbze	64-bit only		
divd	64-bit only	64-bit only	
divd.		64-bit only	

Table 3-44. List of Instructions (continued)

Mnemonic	Book E	PowerPC AIM	e500
divdo	64-bit only	64-bit only	
divdo.		64-bit only	
divdu	64-bit only	64-bit only	
divdu.		64-bit only	
divduo	64-bit only	64-bit only	
divduo.		64-bit only	
divw	x	x	x
divw.	x	x	x
divwo	x	x	x
divwo.	x	x	x
divwu	x	x	x
divwu.	x	x	x
divwuo	x	x	x
divwuo.	x	x	x
eciwx		x	
ecowx		x	
efsabs			SPFPU
efsadd			SPFPU
efscfsf			SPFPU
efscfsi			SPFPU
efscfuf			SPFPU
efscfui			SPFPU
efscmpeq			SPFPU
efscmpgt			SPFPU
efscmplt			SPFPU
efsc tsf			SPFPU
efsc tsi			SPFPU
efsc tsiz			SPFPU
efsc tuf			SPFPU
efsc tui			SPFPU
efsc tuiz			SPFPU
efsddiv			SPFPU
efsmul			SPFPU

Table 3-44. List of Instructions (continued)

Mnemonic	Book E	PowerPC AIM	e500
efsnabs			SPFPU
efsneg			SPFPU
efssub			SPFPU
efststeq			SPFPU
efststgt			SPFPU
efststlt			SPFPU
eieio	replaced with mbar	x	
eqv	x	x	x
eqv.	x	x	x
evabsw			SPE APU
evaddiw			SPE APU
evaddsmiaaw			SPE APU
evaddssiaaw			SPE APU
evaddumiaaw			SPE APU
evaddusiaaw			SPE APU
evaddw			SPE APU
evand			SPE APU
evandc			SPE APU
evcmpeq			SPE APU
evcmpgts			SPE APU
evcmpgtu			SPE APU
evcmplt			SPE APU
evcmpltu			SPE APU
evcntlsw			SPE APU
evcntlzw			SPE APU
evdivws			SPE APU
evdivwu			SPE APU
eveqv			SPE APU
evextsb			SPE APU
evextsh			SPE APU
evfsabs			SPE APU
evfsadd			SPE APU
evfscfsf			SPE APU

Table 3-44. List of Instructions (continued)

Mnemonic	Book E	PowerPC AIM	e500
evfscfsi			SPE APU
evfscfuf			SPE APU
evfscfui			SPE APU
evfscmpeq			SPE APU
evfscmpgt			SPE APU
evfscmplt			SPE APU
evfsctsf			SPE APU
evfsctsi			SPE APU
evfsctsiz			SPE APU
evfsctuf			SPE APU
evfsctui			SPE APU
evfsctui			SPE APU
evfsdiv			SPE APU
evfsmul			SPE APU
evfsnabs			SPE APU
evfsneg			SPE APU
evfssub			SPE APU
evfststeq			SPE APU
evfststgt			SPE APU
evfstslt			SPE APU
evldd			SPE APU
evlddx			SPE APU
evldh			SPE APU
evldhx			SPE APU
evldw			SPE APU
evldwx			SPE APU
evlhhesplat			SPE APU
evlhhesplatx			SPE APU
evlhhosplat			SPE APU
evlhhosplatx			SPE APU
evlhhouplat			SPE APU
evlhhouplatx			SPE APU
evlwhe			SPE APU

Table 3-44. List of Instructions (continued)

Mnemonic	Book E	PowerPC AIM	e500
evlwhex			SPE APU
evlwhos			SPE APU
evlwhosx			SPE APU
evlwhou			SPE APU
evlwhoux			SPE APU
evlwhsplat			SPE APU
evlwhsplatx			SPE APU
evlwwsplat			SPE APU
evlwwsplatx			SPE APU
evmergehi			SPE APU
evmergehilo			SPE APU
evmergelo			SPE APU
evmergelohi			SPE APU
evmhegsmfaa			SPE APU
evmhegsmfan			SPE APU
evmhegsmiaa			SPE APU
evmhegsmian			SPE APU
evmhegumiaa			SPE APU
evmhegumian			SPE APU
evmhesmf			SPE APU
evmhesmfa			SPE APU
evmhesmfaaw			SPE APU
evmhesmfanw			SPE APU
evmhesmi			SPE APU
evmhesmia			SPE APU
evmhesmiaaw			SPE APU
evmhesmianw			SPE APU
evmhessf			SPE APU
evmhessfa			SPE APU
evmhessfaaw			SPE APU
evmhessfanw			SPE APU
evmhessiaaw			SPE APU
evmhessianw			SPE APU

Table 3-44. List of Instructions (continued)

Mnemonic	Book E	PowerPC AIM	e500
evmheumi			SPE APU
evmheumia			SPE APU
evmheumiaaw			SPE APU
evmheumianw			SPE APU
evmheusiaaw			SPE APU
evmheusianw			SPE APU
evmhogsmfaa			SPE APU
evmhogsmfan			SPE APU
evmhogsmiaa			SPE APU
evmhogsmian			SPE APU
evmhogumiaa			SPE APU
evmhogumian			SPE APU
evmhosmf			SPE APU
evmhosmfa			SPE APU
evmhosmfaaw			SPE APU
evmhosmfanw			SPE APU
evmhosmi			SPE APU
evmhosmia			SPE APU
evmhosmiaaw			SPE APU
evmhosmianw			SPE APU
evmhossf			SPE APU
evmhossfa			SPE APU
evmhossfaaw			SPE APU
evmhossfanw			SPE APU
evmhossiaaw			SPE APU
evmhossianw			SPE APU
evmhoumi			SPE APU
evmhoumia			SPE APU
evmhoumiaaw			SPE APU
evmhoumianw			SPE APU
evmhousiaaw			SPE APU
evmhousianw			SPE APU
evmra			SPE APU

Table 3-44. List of Instructions (continued)

Mnemonic	Book E	PowerPC AIM	e500
evmwhsmf			SPE APU
evmwhsmfa			SPE APU
evmwhsmi			SPE APU
evmwhsmia			SPE APU
evmwhssf			SPE APU
evmwhssfa			SPE APU
evmwhumi			SPE APU
evmwhumia			SPE APU
evmwlsmf			SPE APU
evmwlsmfafa			SPE APU
evmwlsmfafaaw			SPE APU
evmwlsmfafanw			SPE APU
evmwlsmi			SPE APU
evmwlsmiaaw			SPE APU
evmwlsmianw			SPE APU
evmwlssf			SPE APU
evmwlssfa			SPE APU
evmwlssfaaw			SPE APU
evmwlssfanw			SPE APU
evmwlssiaaw			SPE APU
evmwlssianw			SPE APU
evmwlumia			SPE APU
evmwlumiaaw			SPE APU
evmwlumianw			SPE APU
evmwlusiaaw			SPE APU
evmwlusianw			SPE APU
evmwsmf			SPE APU
evmwsmfafa			SPE APU
evmwsmfafaaw			SPE APU
evmwsmfafanw			SPE APU
evmwsmi			SPE APU
evmwsmia			SPE APU
evmwsmiaaaw			SPE APU

Table 3-44. List of Instructions (continued)

Mnemonic	Book E	PowerPC AIM	e500
evmwsnian			SPE APU
evmwssf			SPE APU
evmwssfa			SPE APU
evmwssfaa			SPE APU
evmwssfan			SPE APU
evmwumi			SPE APU
evmwumia			SPE APU
evmwumiaa			SPE APU
evmwumian			SPE APU
evnand			SPE APU
evneg			SPE APU
evnor			SPE APU
evor			SPE APU
evorc			SPE APU
evrlw			SPE APU
evrlwi			SPE APU
evrndw			SPE APU
evsel			SPE APU
evslw			SPE APU
evslwi			SPE APU
evsplatfi			SPE APU
evsplati			SPE APU
evsrwis			SPE APU
evsrwiu			SPE APU
evsrws			SPE APU
evsrwu			SPE APU
evstd			SPE APU
evstddx			SPE APU
evstdh			SPE APU
evstdhx			SPE APU
evstdw			SPE APU
evstdwx			SPE APU
evstwhe			SPE APU

Table 3-44. List of Instructions (continued)

Mnemonic	Book E	PowerPC AIM	e500
evstwhex			SPE APU
evstwho			SPE APU
evstwhox			SPE APU
evstwwex			SPE APU
evstwwex			SPE APU
evstwwwo			SPE APU
evstwwox			SPE APU
evsubfsmiaaw			SPE APU
evsubfssiaaw			SPE APU
evsubfumiaaw			SPE APU
evsubfusiaaw			SPE APU
evsubfw			SPE APU
evsubifw			SPE APU
evxor			SPE APU
extsb	x	x	x
extsb.	x	x	x
extsh	x	x	x
extsh.	x	x	x
extsw	64-bit only	64-bit only	
extsw.		64-bit only	
fabs	x	x	
fabs.	x	x	
fadd	x	x	
fadd.	x	x	
fadds	x	x	
fadds.	x	x	
fcfid	x	x	
fcfid.		x	
fcmpo	x	x	
fcmpu	x	x	
fctid	x	x	
fctid.		x	
fctidz	x	x	

Table 3-44. List of Instructions (continued)

Mnemonic	Book E	PowerPC AIM	e500
fctidz.		X	
fctiw	X	X	
fctiw.	X	X	
fctiwz	X	X	
fctiwz.	X	X	
fdiv	X	X	
fdiv.	X	X	
fdivs	X	X	
fdivs.	X	X	
fmadd	X	X	
fmadd.	X	X	
fmadds	X	X	
fmadds.	X	X	
fmr	X	X	
fmr.	X	X	
fmsub	X	X	
fmsub.	X	X	
fmsubs	X	X	
fmsubs.	X	X	
fmul	X	X	
fmul.	X	X	
fmuls	X	X	
fmuls.	X	X	
fnabs	X	X	
fnabs.	X	X	
fneg	X	X	
fneg.	X	X	
fnmadd	X	X	
fnmadd.	X	X	
fnmadds	X	X	
fnmadds.	X	X	
fnmsub	X	X	
fnmsub.	X	X	

Table 3-44. List of Instructions (continued)

Mnemonic	Book E	PowerPC AIM	e500
fnmsubs	x	x	
fnmsubs.	x	x	
fres	x	x	
fres.	x	x	
frsp	x	x	
frsp.	x	x	
frsqrt	x	x	
frsqrt.	x	x	
fsel	x	x	
fsel.	x	x	
fsqrt	x	x	
fsqrt.	x	x	
fsqrts	x	x	
fsqrts.	x	x	
fsub	x	x	
fsub.	x	x	
fsubs	x	x	
fsubs.	x	x	
icbi	x	x	x
icbie	64-bit only		
icblc			Cache locking
icblce			Cache locking (64-bit only)
icbt	x		x
icbte	64-bit only		
icbtls			Cache locking
icbtlse			Cache locking (64-bit only)
isel			Integer select
isync	x	x	x
lbz	x	x	x
lbze	64-bit only		
lbzu	x	x	x
lbzue	64-bit only		
lbzux	x	x	x

Table 3-44. List of Instructions (continued)

Mnemonic	Book E	PowerPC AIM	e500
lbzux	64-bit only		
lbzx	x	x	x
lbzxe	64-bit only		
ld		x	
ldarx		x	
ldarxe	64-bit only		
lde	64-bit only		
ldu		x	
ldue	64-bit only		
ldux		x	
lduxe	64-bit only		
ldx		x	
ldxe	64-bit only		
lfd	x	x	
lfde	64-bit only		
lfdu	x	x	
lfdue	64-bit only		
lfdux	x	x	
lfduxe	64-bit only		
lfdx	x	x	
lfdxe	64-bit only		
lfs	x	x	
lfse	64-bit only		
lfsu	x	x	
lfsue	64-bit only		
lfsux	x	x	
lfsuxe	64-bit only		
lfsx	x	x	
lfsxe	64-bit only		
lha	x	x	x
lhae	64-bit only		
lhau	x	x	x
lhaue	64-bit only		

Table 3-44. List of Instructions (continued)

Mnemonic	Book E	PowerPC AIM	e500
lhaux	x	x	x
lhauxe	64-bit only		
lhax	x	x	x
lhaxe	64-bit only		
lhbrx	x	x	x
lhbrxe	64-bit only		
lhz	x	x	x
lhze	64-bit only		
lhzu	x	x	x
lhzue	64-bit only		
lhzux	x	x	x
lhzuxe	64-bit only		
lhzx	x	x	x
lhzxe	64-bit only		
lmw	x	x	x
lswi	x	x	
lswx	x	x	
lwa		x	
lwarx	x	x	x
lwarxe	64-bit only		
lwaux		x	
lwax		x	
lwbrx	x	x	x
lwbrxe	64-bit only		
lwz	x	x	x
lwze	64-bit only		
lwzu	x	x	x
lwzue	64-bit only		
lwzux	x	x	x
lwzuxe	64-bit only		
lwzx	x	x	x
lwzxe	64-bit only		
mbar	x		x

Table 3-44. List of Instructions (continued)

Mnemonic	Book E	PowerPC AIM	e500
mcrf	x	x	x
mcrfs	x	x	
mcrxr	x	x	x
mcrxr64	64-bit only		
mfapidi	x		
mfcr	x	x	x
mfdcrr	x		
mffs	x	x	
mffs.	x	x	
mfmsr	x	x	x
mfpmr			Performance monitor
mfspir	x	x	x
mfsr		x	
mfsrin		x	
mftrb		x	
msync	x		x
mtcrf	x	x	x
mtdcrr	x		
mtfsb0	x	x	
mtfsb0.	x	x	
mtfsb1	x	x	
mtfsb1.	x	x	
mtfsf	x	x	
mtfsf.	x	x	
mtfsfi	x	x	
mtfsfi.	x	x	
mtmsr	x	x	x
mtmsrd		64-bit only	
mtpmr			Performance monitor
mtspir	x	x	x
mtsr		x	
mtsrd		x	
mtsrdin		x	

Table 3-44. List of Instructions (continued)

Mnemonic	Book E	PowerPC AIM	e500
mtsrin		x	
mulhd	64-bit only	x	
mulhd.		x	
mulhdu	64-bit only	x	
mulhdu.		x	
mulhw	x	x	x
mulhw.	x	x	x
mulhwu	x	x	x
mulhwu.	x	x	x
mulld	64-bit only	x	
mulld.		x	
mulldo	64-bit only	x	
mulldo.		x	
mulli	x	x	x
mulldw	x	x	x
mulldw.	x	x	x
mulldwo	x	x	x
mulldwo.	x	x	x
nand	x	x	x
nand.	x	x	x
neg	x	x	x
neg.	x	x	x
nego	x	x	x
nego.	x	x	x
nor	x	x	x
nor.	x	x	x
or	x	x	x
or.	x	x	x
orc	x	x	x
orc.	x	x	x
ori	x	x	x
oris	x	x	x
rfci	x		x

Table 3-44. List of Instructions (continued)

Mnemonic	Book E	PowerPC AIM	e500
rfi	x	x	x
rfd		x	
rfmci			Machine check
rdcl	64-bit only	x	
rdcl.		x	
rdcr	64-bit only	x	
rdcr.		x	
rdic	64-bit only	x	
rdic.		x	
rdicl	64-bit only	x	
rdicl.		x	
rdicr	64-bit only	x	
rdicr.		x	
rdimi	64-bit only	x	
rdimi.		x	
rlwimi	x	x	x
rlwimi.	x	x	x
rlwinm	x	x	x
rlwinm.	x	x	x
rlwnm	x	x	x
rlwnm.	x	x	x
sc	x	x	x
slbia		x	
slbie		x	
sld	64-bit only	x	
sldi		x	
slw	x	x	x
slw.	x	x	x
srad	64-bit only	x	
srad.		x	
sradi	64-bit only	x	
sradi.		x	
sraw	x	x	x

Table 3-44. List of Instructions (continued)

Mnemonic	Book E	PowerPC AIM	e500
sraw.	x	x	x
srawi	x	x	x
srawi.	x	x	x
srd	64-bit only	x	
srd.		x	
srw	x	x	x
srw.	x	x	x
stb	x	x	x
stbe	64-bit only		
stbu	x	x	x
stbue	64-bit only		
stbux	x	x	x
stbuxe	64-bit only		
stbx	x	x	x
stbx	64-bit only		
std		x	
stdcx.		x	
stdcx.	64-bit only		
stde	64-bit only		
stdu		x	
stdue	64-bit only		
stdux		x	
stduxe	64-bit only		
stdx		x	
stdxe	64-bit only		
stfd	x	x	
stfde	64-bit only		
stfdu	x	x	
stfdue	64-bit only		
stfdux	x	x	
stfduxe	64-bit only		
stfdx	x	x	
stfdxe	64-bit only		

Table 3-44. List of Instructions (continued)

Mnemonic	Book E	PowerPC AIM	e500
stfiwx	x	x	
stfiwx e	64-bit only		
stfs	x	x	
stfse	64-bit only		
stfsu	x	x	
stfsue	64-bit only		
stfsux	x	x	
stfsux e	64-bit only		
stfsx	x	x	
stfsx e	64-bit only		
sth	x	x	x
sthbrx	x	x	x
sthbrx e	64-bit only		
sthe	64-bit only		
sthu	x	x	x
sthue	64-bit only		
sthux	x	x	x
sthux e	64-bit only		
sthx	x	x	x
sthx e	64-bit only		
stmw	x	x	x
stswi	x	x	
stswx	x	x	
stw	x	x	x
stwbrx	x	x	x
stwbrx e	64-bit only		
stwcx.	x	x	x
stwcx. e	64-bit only		
stwe	64-bit only		
stwu	x	x	x
stwue	64-bit only		
stwux	x	x	x
stwux e	64-bit only		

Table 3-44. List of Instructions (continued)

Mnemonic	Book E	PowerPC AIM	e500
stwx	x	x	x
stwx e	64-bit only		
subf	x	x	x
subf.	x	x	x
subfc	x	x	x
subfc.	x	x	x
subfco	x	x	x
subfco.	x	x	x
subfe	x	x	x
subfe.	x	x	x
subfe64	64-bit only		
subfe64o	64-bit only		
subfeo	x	x	x
subfeo.	x	x	x
subfic	x	x	x
subfme	x	x	x
subfme.	x	x	x
subfme64	64-bit only		
subfme64o	64-bit only		
subfmeo	x	x	x
subfmeo.	x	x	x
subfo	x	x	x
subfo.	x	x	x
subfze	x	x	x
subfze.	x	x	x
subfze64	64-bit only		
subfze64o	64-bit only		
subfzeo	x	x	x
subfzeo.	x	x	x
sync	replaced with msync	x	replaced with msync
td	64-bit only	x	
tdi	64-bit only	x	
tlbia		x	

Table 3-44. List of Instructions (continued)

Mnemonic	Book E	PowerPC AIM	e500
tlbie		X	
tlbivax	X		X
tlbivaxe	64-bit only		
tlbre	X		X
tlbsx	X		X
tlbsxe	64-bit only		
tlbsync	X	X	X
tlbwe	X		X
tw	X	X	X
twi	X	X	X
wrtee	X		X
wrteei	X		X
xor	X	X	X
xor.	X	X	X
xori	X	X	X
xoris	X	X	X

Chapter 4

Execution Timing

This chapter describes how the e500 core performs operations defined by instructions and how it reports the results of instruction execution. It gives detailed descriptions of how the core execution units work and how these units interact with other parts of the processor, such as the instruction fetching mechanism, caches register files, and other architected registers. It gives examples of instruction sequences, showing potential bottlenecks and how to minimize their effects. Finally, it includes tables that identify the unit that executes each instruction implemented on the core, the latency for each instruction, and other information useful to assembly language programmers.

4.1 Terminology and Conventions

This section provides an alphabetical glossary of terms used in this chapter. These definitions offer a review of commonly used terms and point out specific ways these terms are used in this chapter.

NOTE

Many of these definitions differ slightly from those used to describe previous processors that implement the PowerPC architecture, in particular with respect to dispatch, issue, finishing, retirement, and write back, so please read this glossary carefully.

- Branch prediction—The process of guessing the direction and target of a branch. Branch direction prediction involves guessing whether a branch will be taken. Target prediction involves guessing the target address of a branch. The e500 does not use the Book E–defined hint bits in the BO operand for static prediction. Clearing BUCSR[BPEN] disables dynamic branch prediction, in which case the e500 predicts every branch as not taken.
- Branch resolution—The determination of whether a branch prediction is correct. If it is, instructions following the predicted branch that may have been speculatively executed can complete (*see* completion). If it is incorrect, the processor redirects fetching to the proper path and purges instructions on the mispredicted path (and any of their results) when the mispredicted branch completes.

- **Complete**—Once an instruction executes and makes its results available for the next instruction, it is eligible to complete. Only instructions in the bottom two entries of the completion queue (CQ) are allowed to complete, and they must complete in order. The completion unit coordinates how instructions (which may have executed out of order) affect architected registers to insure the appearance of program order execution. An instruction completes when it is retired, that is, deleted from the completion queue (CQ). This guarantees that the completed instruction and all previous instructions can cause no exceptions.
- **Decode**—The decode stage determines the issue queue to which each instruction is dispatched (*see* Dispatch) and determines whether the required space is available in both that issue queue and the completion queue. If space is available, it decodes instructions supplied by the instruction queue, renames any source/target operands, and dispatches them to the appropriate issue queues.
- **Dispatch**—Dispatch is the event at the end of the decode stage during which instructions are passed to the issue queues and tracking of program order is passed to the completion queue.
- **Fetch**—The process of bringing instructions from memory (such as a cache or system memory) into the instruction queue.
- **Finish**—An executed instruction finishes by signaling the completion queue that execution has concluded. An instruction is said to be finished (but not complete) when the execution results have been saved in rename registers and made available to subsequent instructions, but the completion unit has not yet updated the architected registers.
- **Issue**—The pipeline stage responsible for reading source operands from rename registers and register files. This stage also assigns and routes instructions to the proper execution unit.
- **Latency**— The number of clock cycles necessary to execute an instruction and make the results of that execution available to subsequent instructions.
- **Pipeline**—In the context of instruction timing, the term ‘pipeline’ refers to the interconnection of the stages. The events necessary to process an instruction are broken into several cycle-length tasks to allow work to be performed on several instructions simultaneously—analogous to an assembly line. As an instruction is processed, it passes from one stage to the next. When the work at one stage is done and the instruction passes to the next stage, another instruction can begin work in the vacated stage.

Although an individual instruction can take many cycles to make results available (*see* Latency), pipelining makes it possible to overlap instruction processing so that throughput (the number of instructions processed per clock cycle) is greater than if pipelining were not implemented.

- **Program order**—The order of instructions in an executing program. More specifically, this term is used to refer to the original order in which program instructions are fetched into the instruction queue from the cache.
- **Rename registers**—Temporary buffers for holding results of instructions that have finished execution but have not completed. The ability to forward results to rename registers allows subsequent instructions to access the new values before they have been written back to the architectural registers.

Note that on the e500, every dispatched instruction is assigned a GPR rename register and a CR field rename register, regardless of whether they have a GPR or CR operand. There is a pair of GPR/CRF rename registers for each completion queue entry, so instructions cannot stall because of a shortage of rename registers or CQ entries.

- **Reservation station**—A buffer between the issue and execute stages that allows instructions to be issued even though the results of other instructions on which the issued instruction may depend are not yet available.
- **Retirement**—Removal of a completed instruction from the completion queue, marking the end of the completion stage. (In other documents, this is often referred to as deallocation.)
- **Speculative instruction**—Any instruction that is currently behind an older branch that has not been resolved.
- **Stage**—Used in two different senses, depending on whether the pipeline is being discussed as a physical entity or a sequence of events. As a physical entity, a stage can be viewed as the hardware that handles operations on an instruction in that part of the pipeline. When viewing the pipeline as a sequence of events, a stage is an element in the pipeline during which certain actions are performed, such as decoding the instruction, performing an arithmetic operation, or writing back the results. Typically, the latency of a stage is one processor clock cycle. Some events, such as dispatch, write-back, and completion, happen instantaneously and may be thought to occur at the end of a stage.

An instruction can spend multiple cycles in one stage; for example, a divide takes multiple cycles in the execute stage.

An instruction can also be represented in more than one stage simultaneously, especially in the sense that a stage can be seen as a physical resource. For example, when instructions are dispatched, they are assigned a place in the CQ at the same time they are passed to the issue queues.

- **Stall**—An occurrence when an instruction cannot proceed to the next stage. A delay initiated to resolve a data or resource hazard, that is, a situation in which a planned instruction cannot execute in the proper clock cycle because data or resources needed to execute the instruction are not yet available.

- **Superscalar**—A superscalar processor is one that can issue multiple instructions concurrently from a conventional linear instruction stream. In a superscalar implementation, multiple instructions can execute in parallel at the same time.
- **Throughput**—The number of instructions that are processed per cycle. In particular, throughput describes the performance of a multiple-stage pipeline where a sequence of instructions may pass through with a throughput that is much faster than the latency of an individual instruction. For example, in the four-stage multiple-cycle pipeline (MU), a series of **mulli** instructions has a throughput of one instruction per clock cycle even though it takes 4 cycles for an individual **mulli** instruction to execute.
- **Write-back**—Write-back (in the context of instruction handling) occurs when a result is written into the architecture-defined registers (typically the GPRs). On the e500, write-back occurs in the clock cycle after the completion stage. Results in the write-back buffer cannot be flushed. If an exception occurs, results from previous instructions must write back before the exception is taken.

4.2 Instruction Timing Overview

The e500 design minimizes the number of clock cycles it takes to fetch, decode, dispatch, issue, and execute instructions and to make the results available for a subsequent instruction. To improve throughput, the e500 implements pipelining, superscalar instruction issue, and multiple execution units that operate independently and in parallel.

Figure 4-1 shows the path that instructions take through the seven stages of the e500 master pipeline: two fetch stages, decode/dispatch, issue, execute, complete, and write-back stages. The LSU and MU execution units are also multiple-stage pipelines.

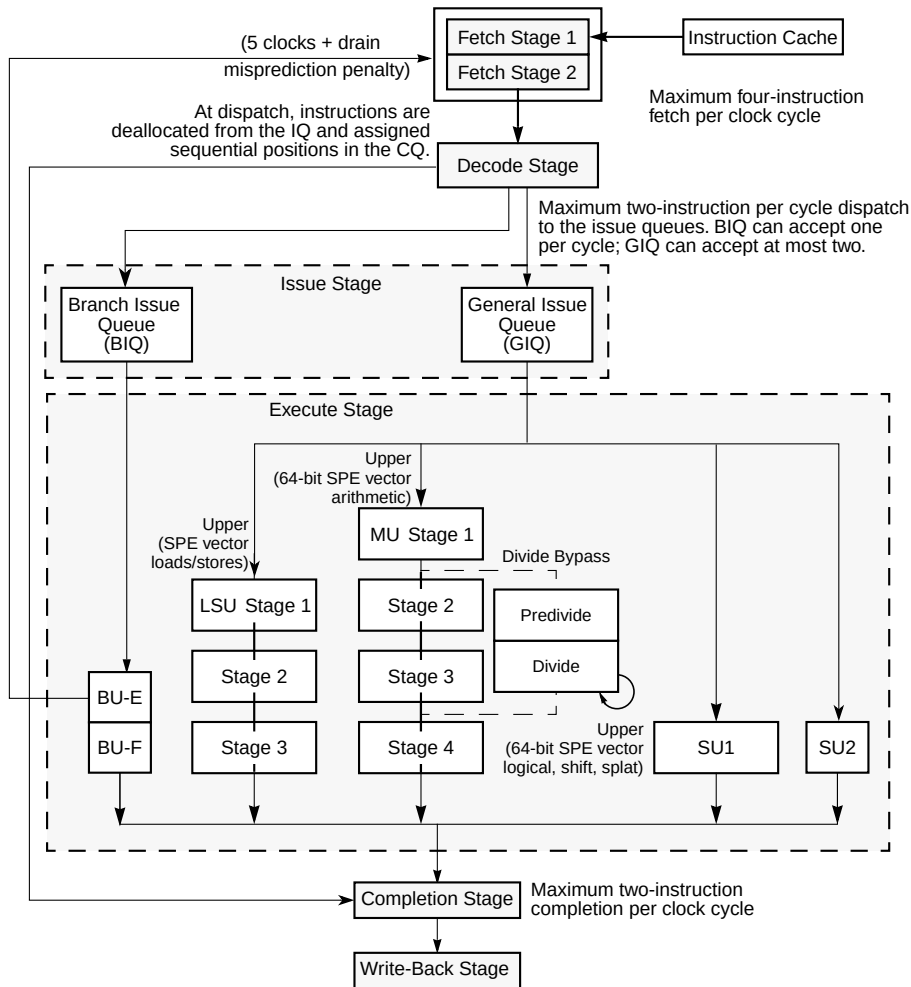


Figure 4-1. Instruction Flow Pipeline Diagram Showing Pipeline Stages

Some instructions, such as loads and stores, access memory and require additional clock cycles between the execute phase and the write-back phase. These latencies vary depending on whether the access is to cacheable or noncacheable memory, whether it hits in the L1 cache, whether the cache access generates a write-back to memory, whether the access causes a snoop hit from another device that generates additional activity, and other conditions that affect memory accesses.

The e500 can complete as many as two instructions on each clock cycle.

Instruction Timing Overview

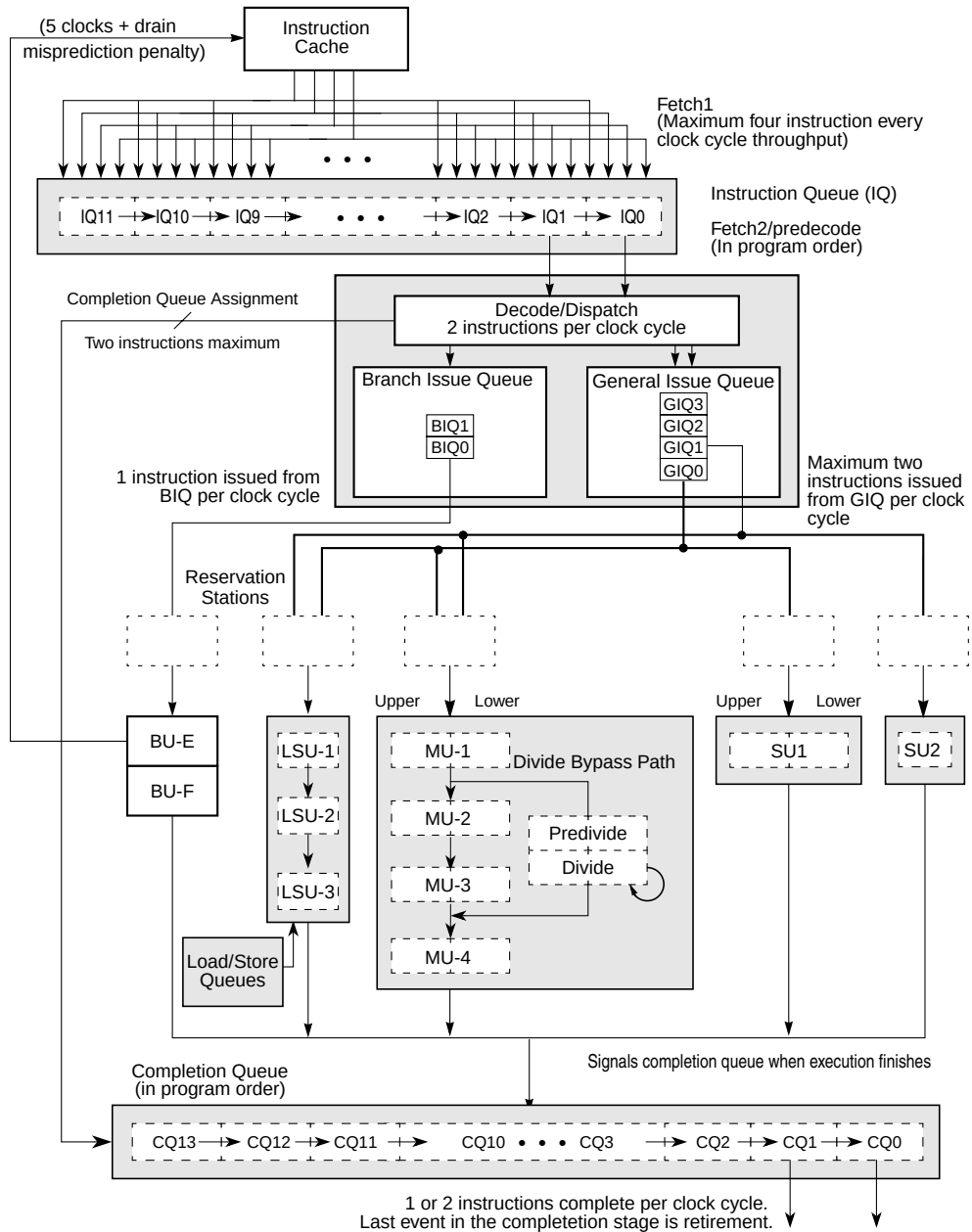


Figure 4-2. e500 Instruction Flow Diagram or Superscalar/Pipeline

The instruction pipeline stages are described as follows:

- **Instruction fetch**—Includes the clock cycles necessary to request an instruction and the time the memory system takes to respond to the request. Instructions retrieved are latched into the instruction queue (IQ) for subsequent consideration by the dispatcher.

The fetcher tries to initiate a fetch in every cycle in which it is guaranteed that the IQ has room for fetched instructions. Instructions are typically fetched from the L1 instruction cache; if caching is disabled, instructions are fetched from the instruction cache line fill buffer, shown in Figure 4-8. Likewise, on a cache miss, as many as four instructions can be forwarded to the fetch unit from the line fill buffer as the cache line is passed to the instruction cache.

Fetch timing is affected by many things, such as whether an instruction is in the on-chip instruction cache or an L2 cache (if implemented). Those factors increase when it is necessary to fetch instructions from system memory and include the processor-to-bus clock ratio, the amount of bus traffic, and whether any cache coherency operations are required.

Fetch timing is also affected by whether the effective address translation is available in a TLB, as described in 4.3.2.1, “L1 and L2 TLB Access Times.

Because there are so many variables, unless otherwise specified, the instruction timing examples in this chapter assume optimal performance and show the portion of the fetch stage in which the instruction is in the instruction queue. The two fetch stages are primarily involved in retrieving instructions.

- The **decode/dispatch** stage fully decodes each instruction; most instructions are dispatched to the issue queues (**isync**, **rfi**, **sc**, **nops**, and some other instructions do not go to issue queues). Note that on the e500, every dispatched instruction is assigned a GPR rename register and a CR field rename register, regardless of whether they specify a GPR or CR operand. There is a pair of GPR/CRF rename registers for each CQ entry (even for instructions that do not access the CR or GPRs), so instructions cannot stall because of a shortage of rename registers or CQ entries.

The two issue queues, BIQ and GIQ, can accept as many as one and two instructions, respectively, in a cycle. Instruction dispatch requires the following:

- Instructions dispatch only from the two lowest IQ entries—IQ0 and IQ1.
- A total of two instructions can be dispatched to the issue queues per clock cycle.
- Space must be available in the CQ for an instruction to decode and dispatch (this includes instructions that are assigned an entry in the CQ but not in an issue queue).

In this chapter, dispatch is treated as an event at the end of the decode stage. Dispatch dependencies are described in Section 4.7.2, “Dispatch Unit Resource Requirements.”

- The issue stage reads source operands from rename registers and register files and determines when instructions are latched into reservation stations.

The general behavior of the two issue queues is described as follows:

- The GIQ accepts as many as two instructions from the dispatch unit per cycle. SU1, SU2, MU, and all LSU instructions (including SPE APU loads and stores) are dispatched to the GIQ, shown in Figure 4-3.

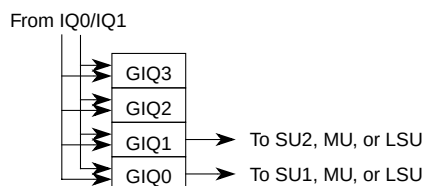


Figure 4-3. GPR Issue Queue (GIQ)

Instructions can be issued out-of-order from the bottom two GIQ entries (GIQ1–GIQ0). GIQ0 can issue to SU1, MU, and LSU. GIQ1 can issue to SU2, MU, and LSU.

Note that SU2 executes a subset of the instructions that can be executed in SU1. The ability to identify and dispatch instructions to SU2 increases the availability of SU1 to execute more computationally intensive instructions.

An instruction in GIQ1 destined for SU2 or the LSU need not wait for an MU instruction in GIQ0 that is stalled behind a long-latency divide.

- The execute stage accepts instructions from its issue queue when the appropriate reservation stations are not busy. In this stage, the operands assigned to the execution stage from the issue stage are latched.

The e500 contains the following execution units:

- Branch unit (BU)—executes branches and CR logical operations
- Load/store unit (LSU)—executes loads from and stores to memory, as well as some MMU control, cache control, and cache locking instructions. This includes byte, half-word, and word instructions defined by the PowerPC architecture and 64-bit load and store instructions defined as part of the SPE APU. Load/store queues are described in Section 4.4.2, “Load/Store Unit Queueing Structures.”
- Two simple units (SU1 and SU2)—execute all move to/from special-purpose register (SPR) instructions, logical instructions, and all computational instructions except multiply and divide instructions. Most instructions take one cycle to execute in either of these units. These execution units also execute all vector and scalar computational instructions (except multiply and divide instructions) defined by the SPE APU, as follows:

- SU1 executes both 32- and 64-bit SPE logical instructions, simple integer arithmetic, and bit manipulation instructions, such as merges, shifts, and splats.
- SU2 executes a subset of the instructions that can be executed in SU1. These include **brinc** and the 32-bit SPE scalar floating-point logical instructions, **efsabs**, **efsnabs**, **efsneg**, **efststeg**, **efststgt**, and **efststlt**.

Table 4-6 identifies which Book E instructions execute in SU1 and SU2 and shows their latencies; Table 4-8 identifies which SPE APU instructions execute in SU1 and SU2 and shows their latencies. Note that most SU instructions execute in 1 cycle, while some instructions (such as **mtspr** and **mfspr**) take longer.

- Multiple-cycle IU (MU), sometimes referred to as the complex instruction unit, executes integer multiplication and division instructions, and addition, subtraction, multiplication, and division for all vector and scalar SPE instructions.

NOTE

As suggested by Figure 4-1, SPE APU instructions execute in two single-wide (32-bit) execution units which together form double-wide (64-bit) results for SPE APU instructions. For non-SPE APU instructions, only the 32-bit lower-unit results are written back to architected registers.

The execution unit executes the instruction (perhaps over multiple cycles), writes results on its result bus, and notifies the CQ when the instruction finishes. The execution unit reports any exceptions to the completion stage. Instruction-generated exceptions are not taken until the excepting instruction is next to retire.

Most integer instructions have a 1-cycle latency, so results of these instructions are available 1 clock cycle after an instruction enters the execution unit. The LSU and MU are pipelined, as shown in Figure 4-4.

- The complete and write-back stages maintain the correct architectural machine state and commit results to the architecture-defined registers in the proper order. If completion logic detects an instruction containing an exception status or a mispredicted branch, all following instructions are cancelled, their execution results in rename registers are discarded, and the correct instruction stream is fetched.

The complete stage ends when the instruction is retired. Two instructions can be retired per clock cycle. If no dependencies exist, as many as two instructions are retired in program order. Section 4.7.4, “Completion Unit Resource Requirements” describes completion dependencies.

The write-back stage occurs in the clock cycle after the instruction is retired.

Conventions used in the instruction timing examples are as follows:

Instruction Timing Overview

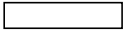
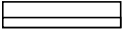
	Fetch —Instructions are fetched from memory and placed in the 12-entry IQ. The latency associated with accessing an instruction depends on whether the instruction is in the on-chip cache or system memory (in which case latency is further affected by bus traffic, bus clock speed, and address translation issues). Therefore, in the examples in this chapter, the diagrams and fetch stage shown is for the common case of instructions hitting in the instruction cache.
	Decode —As many as two eligible instructions dispatch from IQ0–IQ1 to the appropriate issue queue. Note that isync , rfl , sc , and some other instructions do not go to issue queues. At the same time, the instruction is assigned an entry in the completion queue.
	Issue —Instructions are dispatched to issue queues from the instruction queue entries. At the end of the issue stage, instructions and their operands, if available, are latched into execution unit reservation stations. The black stripe is a reminder that the instruction occupies an entry in the CQ, described in Figure 4-4.
	Execute —The operations specified by an instruction are being performed by the appropriate execution unit. The black stripe is a reminder that the instruction occupies an entry in the CQ, described in Figure 4-4.
	Complete —Execution has finished. When all completion requirements are met, the instruction is retired from the CQ. The results are written back to architecture-defined registers in the clock cycle after retirement.
	Write back —The instruction has retired and its results are written back to the architecture-defined registers.

Figure 4-4 shows the relationships between stages and events associated with them.

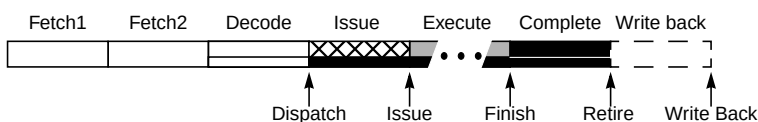


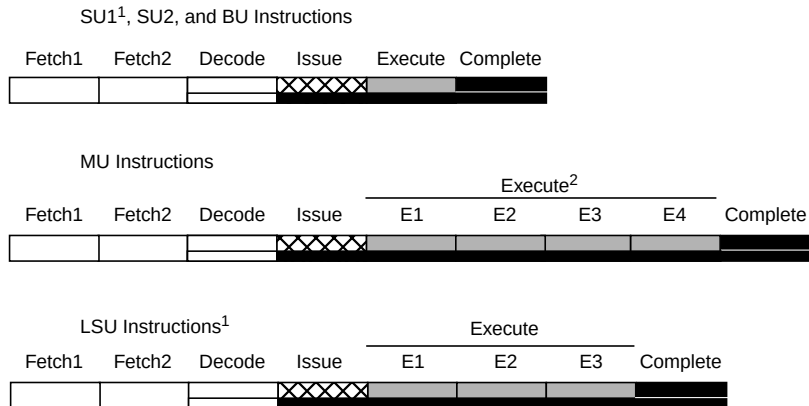
Figure 4-4. Execution Pipeline Stages and Events

The events are described as follows:

- **Dispatch** (at the end of decode)—An instruction is dispatched to the appropriate issue queue at the end of the decode stage. At dispatch, the instruction passes to the issue pipeline stage by taking a place in the completion queue and in one of the two issue queues.
- **Issue** (at the end of the issue stage)—The issue stage ends when the instruction is issued to the appropriate execution unit.

- **Finish** (at the end of the execute stage)—An instruction finishes when the CQ is signalled that execution results are available to subsequent instructions. Architecture-defined registers are not updated until the instruction is retired.
- **Retire** (at the end of the complete stage)—An instruction retires when it has updated architecture-defined registers with its results and is removed from the completion queue.
- **Write back** (at the end of the write-back stage)—The results of a retired instruction are written back to the architecture-defined register.

Figure 4-5 shows the stages of e500 execution units.



¹Execution-serialized instructions require additional execution cycles (not shown).

²Divide instruction may take 4, 11, 19, 29, or 35 clock cycles to complete.

Figure 4-5. Execution Stages

4.3 General Timing Considerations

When the fetch pipeline is full, as many as four instructions can be fetched to the IQ during each clock cycle. Two instructions per clock cycle can be dispatched to the issue queues. Two instructions from the GIQ and one instruction from the BIQ can issue per clock cycle to the appropriate execution units. Two instructions can retire per cycle, and two instructions can write back per cycle.

The e500 executes multiple instructions in parallel, using hardware to manage dependencies. When an instruction is issued, the availability of the GPR or rename registers send the source data to the appropriate reservation station. Register files and rename registers have sufficient bandwidth to allow dispatch of two instructions per clock cycle.

The BU executes branches received from the BIQ. Note that branch prediction is performed in parallel with the fetch stages. If the BU resolves a prediction as incorrect, the following steps are taken:

1. Fetch is redirected, and any instructions from the incorrect path are purged from the pipeline.
2. The mispredicted branch is marked as such in the CQ.
3. Eventually, the branch is retired and the CQ, issue queue, and execution units are flushed. If the correct-path instructions reach the IQ before the back half of the pipeline is flushed, they stall in the IQ until the flush occurs.

After an instruction executes, results are made available to subsequent instructions in the appropriate rename registers. The architecture-defined GPRs are updated during the write-back stage. Branch instructions that update the LR or CTR write back their results in a similar fashion.

After instruction execution, results are made available to subsequent instructions in the appropriate rename registers. If a subsequent instruction needs the result as a source operand, the result is simultaneously made available to the appropriate execution unit, which allows a data-dependent instruction to be decoded and dispatched without waiting to read the data from the architected register file. Results are then stored into the correct architected GPR during the write-back stage. Branch instructions that update either the LR or CTR write back their results in a similar fashion.

Section 4.3.1, “General Instruction Flow,” describes this process.

4.3.1 General Instruction Flow

To resolve branch instructions and improve the accuracy of branch predictions, the e500 implements a dynamic branch prediction mechanism using a 512-entry branch target buffer (BTB), a four-way set associative address cache of branch target addresses. An entry is allocated in the BTB whenever a branch resolves as taken—branches that have not been allocated are always predicted as not taken. Each BTB entry holds a 2-bit branch history table, whose value is incremented or decremented depending on whether the branch was taken. These bits can take four values: strongly taken, weakly taken, weakly not taken, and strongly not taken.

When the branch is resolved (in the branch execute stage), the value of these bits is updated, unless a strong prediction was correct. So if a branch whose counter indicates weakly taken is resolved as taken, the bits are incremented so that the prediction becomes strongly taken. If the branch resolves as not taken, the prediction changes to weakly not-taken.

The e500 does not implement the static branch prediction that is defined by the PowerPC architecture. The BO prediction in branch encodings is ignored.

Dynamic branch prediction is enabled by setting BUCSR[BPEN]. Clearing BUCSR[BPEN] disables dynamic branch prediction, in which case the e500 predicts every branch as not taken.

- Link register—The e500 also avoids stalls by implementing a branch link register. A single **bclr**/branch-and-link pair can be stored and the **bclr** target address can be predicted from the link register rather than requiring a stall until the **lwz/mtlr** subroutine restore sequence completes.

Branch instructions are treated like any other instruction and are assigned CQ entries to ensure that the CTR and LR are updated sequentially.

The dispatch rate is affected by the serializing behavior of some instructions and on the availability of issue queues, execution units, rename registers, and CQ entries. Instructions are dispatched in program order; an instruction in IQ1 cannot be dispatched ahead of one in IQ0.

4.3.2 Instruction Fetch Timing Considerations

Instruction fetch latency depends on the following factors:

- Whether the page translation for the effective address of an instruction fetch is in a TLB. This is described in Section 4.3.2.1, “L1 and L2 TLB Access Times.”
- If the page translation is not in a TLB, an instruction TLB miss interrupt is taken. Other conditions may also cause an instruction fetch to take an interrupt, as described in Section 4.3.2.2, “Interrupts Associated with Instruction Fetching.” General interrupt latency and pipeline behavior is described in Section 4.3.4, “Interrupt Latency.”
- Whether the fetch hits the L1 instruction cache. If a cache miss occurs, a memory transaction is required in which fetch latency is affected by bus traffic and bus clock speed. These issues are discussed further in the Section 4.3.2.3, “Cache-Related Latency.”

4.3.2.1 L1 and L2 TLB Access Times

The L1 TLB arrays are checked for a translation hit in parallel with the on-chip L1 cache lookups and incur no penalty on an L1 TLB hit. If the L1 TLB arrays miss, the access proceeds to the L2 TLB arrays. For L1 instruction address translation misses, the L2 TLB latency is at least 5 clocks; for L1 data address translation misses, the L2 TLB latency is at least 6 clocks. These access times may be longer, depending on arbitration performed by the L2 arrays for simultaneous instruction L1 TLB misses, data L1 TLB misses, the execution of TLB instructions, and TLB snoop operations (snooping of TLBINV operations on the CCB).

Note that when a TLBINV operation is detected on the CCB, the L2 MMU arrays become inaccessible due to the snooping activity caused by the TLBINV.

If the MMU is busy due to a higher priority operation, such as a **tlbivax**, instructions cannot be fetched until that operation completes.

If the page translation is in neither TLB, an instruction TLB error interrupt occurs, as described in Section 5.7.13, “Instruction TLB Error Interrupt.”

TLBs are described in detail in Chapter 13, “Memory Management Units.”

4.3.2.2 Interrupts Associated with Instruction Fetching

An instruction fetch can generate the following interrupts:

- An instruction TLB error interrupt occurs when the effective address translation for a fetch is not found in the TLBs. This interrupt is described in detail in Section 5.7.13, “Instruction TLB Error Interrupt”
- An instruction storage exception is caused when one of the following occurs during an attempt to fetch instructions:
 - An execute access control exception is caused when one of the following conditions exist:
 - In user mode, an instruction fetch attempts to access a memory location that is not user mode execute enabled (page access control bit UX = 0).
 - In supervisor mode, an instruction fetch attempts to access a memory location that is not supervisor mode execute enabled (page access control bit SX = 0).
 - A byte ordering exception may occur when the implementation cannot fetch the instruction in the byte order specified by the page’s endian attribute. On the e500, accesses that cross a page boundary such that endianness changes causes a byte ordering exception. .

When an instruction storage interrupt occurs, the processor suppresses execution of the instruction causing the exception. For more information, see Section 5.7.4, “Instruction Storage Interrupt.”

4.3.2.3 Cache-Related Latency

When the instruction fetcher requests instructions from the instruction cache, the following may happen:

- If the instruction fetch hits the instruction cache, it takes only two clock cycles after the request for as many as four instructions to enter the IQ. Note that the cache is not blocked to internal accesses during a cache reload (hits under misses).

Note that the cache allows a hit under one miss and is only blocked by a cache line reload for the cycle when the cache write happens. For example, if a cache miss is

discarded by a misprediction and a new fetch hits, the cache allows instructions to come back. As many as four instructions per cycle are fetched from the cache until the original miss comes back and a cache reload is performed, which blocks the cache for 1 cycle.

If the cache is busy due to a higher priority operation, such as an **icbi** or a cache line reload, instructions cannot be fetched until that operation completes.

- If an instruction fetch misses the on-chip instruction cache, the e500 initiates a core complex bus transaction to the non-core memory system.

To minimize the effect of bus contention, the Book E architecture defines WIM bits that define caching characteristics for the corresponding page. Accesses to caching-inhibited memory locations never update the L1 caches.

If a cache-inhibited access hits in the cache, the cache block is invalidated. If the cache block is marked modified, it is copied back to memory before being invalidated. Where caching is permitted, memory is configured as either write-back or write-through, as described in Section 12.3.4, “WIMGE.”

4.3.3 Dispatch, Issue, and Completion Considerations

The core’s ability to dispatch as many as two instructions per cycle depends on the mix of instructions and on the availability of issue queues, destination rename registers, and CQ entries. Several of these factors are shown in the previous instruction timing examples.

Although as many as two instructions can be dispatched in parallel, they cannot be dispatched out of order; that is, an instruction in IQ1 cannot be dispatched unless the instruction in IQ0 can as well.

Note that instructions can issue out of order from GIQ0 and GIQ1. GIQ0 can issue to SU1, MU, and LSU. GIQ1 can issue to SU2, MU, and LSU. If the instruction in GIQ0 is stalled (reservation station busy), the instruction in GIQ1 can issue if its reservation station is available.

All issued instructions pass through a single-entry reservation station associated with each execution unit. If a data dependency keeps an instruction from starting execution, that instruction is held in a reservation station. Execution begins during the same clock cycle that the rename register is updated with the data the instruction is dependent on.

The issue queues and reservation stations allow the e500 to dispatch decoded instructions even if execution units are busy. The dispatcher also locates source operands and assigns rename registers for destination operands. The issue logic reads operands from register files and rename registers and routes instructions to the proper execution unit. Execution begins when all operands are available, the instruction is in the reservation station, and any execution serialization requirements are met.

The CQ maintains program order after instructions are dispatched, guaranteeing in-order completion and a precise exception model. Instruction state and other information required for completion are kept in this 14-entry FIFO. All instructions complete in order; none can retire ahead of a previous instructions. In-order completion ensures the correct architectural state when the e500 must recover from a mispredicted branch or exception.

Instructions are retired much as they are dispatched. As many as two instructions can be retired simultaneously, but instructions cannot be retired out of order. Note the following:

- Instructions must be non-speculative to complete.
- As many as two rename registers can be updated per clock cycle. Load and store with update instructions, such as **lwzu**, require two rename registers and are broken into two instructions at dispatch (**lwzu** is broken into **lwz** and **addi**). As described in Section 4.3.3.1, “GPR and CR Rename Register Operation,” these two instructions are assigned two CQ entries and are each assigned a pair of CR and GPR renames at dispatch.
- Some instructions have retirement restrictions, such as retiring only out of CQ0.

Program-related exceptions are signaled when the instruction causing the exception reaches CQ0. Previous instructions are allowed to complete before the exception is taken, which ensures that any exceptions those instructions may cause are taken.

4.3.3.1 GPR and CR Rename Register Operation

To avoid contention for a given register file location during out-of-order execution, the e500 provides 14 rename registers for holding instruction results before the completion commits them to the architecture-defined registers. In addition to the 14 GPR renames, the e500 also provides fourteen 4-bit CR field renames. Because the number of renames is the same as the number of completion queue entries and each completion queue entry is guaranteed to require at most one rename of each type, the e500 (unlike earlier implementations) cannot run out of renames unless it has also run out of CQ entries.

When an instruction is decoded, a rename register is allocated for the results of that instruction. Each instruction is allocated one GPR rename register and one CR field rename register. Results from rename registers are transferred to the architecture-defined registers in the write-back stage, at which point renames are deallocated.

If branch prediction is incorrect, instructions after the branch are flushed from the CQ and any results of those instructions are flushed from the rename registers.

Load and store with update instructions require multiple target GPRs and so are broken into two instructions in decode stage; each piece is assigned one rename of each type. If an instruction is issued to a reservation station because of a data dependency, issue logic provides a tag to the execution unit identifying the rename register that forwards required data at completion. Execution can begin when source data reaches the rename register.

An **lwzu** is essentially split into an **lwz** and an **addi**, but takes up one GIQ entry; however, the two parts can issue in different cycles, as shown in Figure 4-6.

	Clock 1	Clock 2	Clock 3	Clock 4	Clock 5	Clock 6
IQ11						
.	.	.	.	.	.	.
.	.	.	.	.	.	.
IQ6						
IQ5	I5					
IQ4	I4					
IQ3	lwzu	I5				
IQ2	I2	I4				
IQ1	I1	lwzu				
IQ0	I0	I2				
BIQ1						
BIQ0						
GIQ3			I5			
GIQ2			I4	I5		
GIQ1		I1	lwz/addi	I4		
GIQ0		I0	I2	lwz	I5	
CQ13						
.	.	.	.	.	.	.
.	.	.	.	.	.	.
CQ6				I5		
CQ5				I4		
CQ4			addi	addi (SU2)		I5 (SU1)
CQ3			lwz	lwz	I4 (SU2)	I4
CQ2			I2	I2 (LSU)	addi	addi
CQ1		I1	I1 (SU2)	I1 ✓	lwz (LSU)	lwz (LSU)
CQ0		I0	I0 (SU1)	I0 ✓	I2 (LSU)	I2 (LSU)

The ✓ indicates that the instruction has finished execution and can be retired.

Figure 4-6. Splitting of an lwzu Instruction into Two Operations

The **lwzu** is dispatched from IQ1 at the end of clock cycle 2. After dispatch, in clock 3, the **lwzu** is assigned two positions (an **lwz** and an **addi**) in the CQ, and a pair of CR field and GPR renames is assigned to each; however, the two component instructions occupy a single GIQ entry (GIQ1). Instruction 1 (a load) is in GIQ0. Because instruction 1 claims the LSU in clock cycle 4, the **lwz** cannot be issued; however, the update (**addi**) issues to SU2. The **lwz** drops to GIQ0. Issue logic remembers that the update (**addi**) has been issued, so it checks only for LSU availability. In clock 5, the **lwz** issues to the LSU.

4.3.3.2 LR and CTR Shadow (Speculative) Registers

The decode stage manages one speculative copy each of the LR and of the CTR. This allows one-level deep speculation for branch-to-LR and branch-to-CTR instructions.

4.3.3.3 Instruction Serialization

Although the e500 core can dispatch and complete two instructions per cycle, serializing instructions limit dispatch and completion to one instruction per cycle. There are six basic types of instruction serialization:

- Presync serialization—Presync-serialized instructions are held in the instruction queue until all prior instructions have completed. They are then decoded and execute. For example, instructions such as **mfsprr** that read a non-renamed status register are marked as presync-serialized.
- Postsync serialization—Postsync-serialized instructions, such as **mtspr[XER]**, prevent other instructions from decoding until the serialized instruction completes. For example, instructions that modify processor state in a way that affects the handling of future instruction execution are marked with postsync-serialization. These instructions are identified in the latency tables in Section 4.6, “Instruction Latency Summary.”
- Move-from serialization—Move-from serialization is a weaker synchronization than presync serialization. A move-from serialized instruction can decode, but will stall in an execution unit’s reservation station until all prior instructions have completed. If the instruction is currently in the reservation station and is the oldest instruction, it can begin execution in the next cycle. Note that subsequent instructions can decode and execute while a move-from serialized instruction is pending. Only **mfcrr** and **mfxer** are move-from serialized, so that they do not examine architectural state until all older instructions that could affect the architectural state have completed.
- Move-to serialization—Move-to serialization is a weaker synchronization than move-from serialization. If the instruction is currently in the reservation station or is currently being issued to the reservation station/execution unit, and is the oldest instruction, it can begin execution in the next cycle. Most **mtspr** and **mfsprr** instructions are move-to serialized.
- Refetch serialization—Refetch-serialized instructions force refetching of subsequent instructions after completion. Refetch serialization is used when an instruction has changed or may change a particular context needed by subsequent instructions. Examples include **isync**, **sc**, **rfi**, **mtspr[XER]**, and any instruction that toggles the summary-overflow (SO) bit.
- Store serialization (applicable to stores and some LSU instructions that access the data cache)—Store-serialized instructions are dispatched and held in the LSU’s finished store queue. They are not committed to memory until all prior instructions have completed. While a store-serialized instruction waits in the finished store queue, other load/store instructions can be freely executed. Some store-serialized instructions are further restricted to complete only from CQ0. Only one store-serialized instruction can complete per cycle, although non-serialized

instructions can complete in the same cycle as a store-serialized instruction. In general, all stores and cache operation instructions are store serialized.

4.3.4 Interrupt Latency

The core complex interrupt latency is defined here as the number of core clocks between the sampling of the interrupt signal as asserted and the fetch of the first instruction in the handler. The interrupt latency of the core complex is determinate unless a guarded load or a cache-inhibited **stwcx.** instruction is being executed. As a guarded load or a cache-inhibited **stwcx.** instruction is being executed, the interrupt latency of the core complex is indeterminate. The minimum interrupt latency for the core complex is 3 core clocks and the maximum is 8. These clocks are described in Table 4-1.

Table 4-1. Interrupt Latency

Action	Guarded Window	Reaching a Recoverable State.	Interrupt Processing	Redirect
Clocks	4/0	2/1	1	1

The interrupt latency defined in Table 4-1 does not include the 2 bus clock cycles required to synchronize the interrupt signal from the pad.

The guarded window is required to ensure that a nonspeculative load or store instruction (such as a **stwcx.** or guarded load) is not issued to the external bus by the load/store unit and then flushed by the completion unit. The guarded window is invoked if and only if a load or store instruction is the oldest instruction being executed (in CQ0). If an asynchronous interrupt is being serviced and the oldest instruction is not a load/store instruction, the core complex goes straight from sampling the interrupt to ensuring a recoverable state and issuing an exception. If a load/store instruction is oldest, the core complex waits 4 clocks before ensuring a recoverable state. During this time, any instruction finished by the load/store unit is deallocated.

NOTE

Guarded **lmw** and **stmw** instructions can be interrupted before the instruction is completed and then restarted after the interrupt is serviced.

4.3.5 Memory Synchronization Instruction Timing Considerations

The **msync** instruction provides a memory barrier throughout the memory hierarchy. On the e500, this instruction waits for any preceeding data memory accesses to reach the point of coherency (that is, visible to the entire memory hierarchy), then it is broadcast on the e500 bus. The **msync** completes only after its address tenure is successfully performed on

the bus (without being ARTRYed). Subsequent instructions may execute out-of-order, but complete only after **msync** completes.

The e500 implements two variations of **mbar**, as determined by the value of the MO field (bits 6–10) of the instruction, as follows:

- When MO = 0, **mbar** behaves identically to **msync**.
- When MO = 1, **mbar** is a weaker, faster memory barrier. It ensures that all data accesses caused by previous instructions complete before those caused by any subsequent instructions. This order is seen by all mechanisms. However, unlike **msync** and **mbar** with MO = 0, **mbar** with MO = 1 does not wait for its address tenure to be performed successfully on the bus before allowing subsequent instructions to complete. This provides a faster way to order data accesses.

4.4 Execution Unit Timings

The following sections describe instruction timing considerations within each of the respective execution units in the e500.

4.4.1 Branch Unit Execution Timing

Flow control operations (conditional branches, unconditional branches, and traps) may disrupt normal instruction flow, and when program flow changes, the IQ must be reloaded with the target instruction stream. Previously issued instructions continue executing while the new instruction stream makes its way into the IQ, but depending on whether the target instruction is in the L1 instruction cache (or L2 cache if implemented) or in system memory, opportunities may be missed to execute instructions.

The penalties associated with e500 flow control operations are minimized by performance features such as dynamic branch prediction, a link register, and nonblocking caches. Timing for branch instruction execution is affected by whether the following occur:

- The branch is taken.
- The target instruction stream is in the on-chip cache.
- The branch is predicted.
- The prediction is correct.

4.4.1.1 Branch Instructions and Completion

Branch instructions are not folded on the e500, all branch instructions receive a CQ entry at dispatch and must write back in program order.

Branch instructions are dispatched to the BIQ and are assigned a CQ slot, as shown in Figure 4-7.

	Clock 1	Clock 2	Clock 3	Clock 4	Clock 5
IQ11					
.	.	.	.	.	.
.	.	.	.	.	.
.	.	.	.	.	.
IQ6					
IQ5					
IQ4	add3				
IQ3	add2				
IQ2	add1	add3			
IQ1	bc	add2			
IQ0	cmp	add1	add3		
BIQ1					
BIQ0		bc			
GIQ3					
GIQ2					
GIQ1			add2		
GIQ0		cmp	add1	add3	
CQ13					
.	.	.	.	.	.
.	.	.	.	.	.
.	.	.	.	.	.
CQ6					
CQ5					
CQ4					
CQ3			add2	add2 (SU2)	add3 (SU1)
CQ2			add1	add1 (SU1)	add2√
CQ1		bc	bc (BU)	bc (BU)	add1√
CQ0		cmp	cmp (SU1)	cmp√	bc√

The √ indicates that the instruction has finished execution and no longer occupies an execution unit.

Figure 4-7. Branch Completion (LR/CTR Write-Back)

In this example, the **bc** depends on **cmp** and is predicted as not taken. At the end of clock cycle 1, the **cmp** and **bc** are dispatched to the GIQ and BIQ, respectively, and are issued to SU1 and the BU at the end of clock 2.

In clock cycle 3, the **cmp** executes in SU1 but the **bc** cannot resolve and complete until the **cmp** results are available; add1 and add2 are dispatched to the GIQ.

In cycle 4, the **bc** resolves as correctly predicted; add1 and add2 are issued to the SUs and are marked as nonspeculative, and add3 is dispatched to the GIQ. The **cmp** is retired from the CQ at the end of cycle 4.

In cycle 5, the **bc**, add1, and add2 finish execution, and **bc** and add1 retire from the CQ.

4.4.1.2 Branch Prediction and Resolution

The e500 dynamic branch prediction mechanism differs from its predecessors in that branches are detected and predicted earlier, in the two fetch stages. This processor-specific

mechanism is implemented in hardware that monitors branch instruction behavior and maintains a record from which the next occurrence of the branch instruction is predicted.

The e500 does not support static branch prediction—the BO prediction in branch instructions is ignored.

When a branch instruction enters the instruction pipeline for the first time, it is predicted as not taken. If the branch is not taken, nothing is allocated in the BTB. However, if the branch is taken, the misprediction causes an entry for this branch to be allocated in the BTB. The initial prediction is strongly taken.

The BPU detects whether a fetch group includes any branches that hit in the BTB, and if so, determines the fetching path based on the prediction and the target address.

If the prediction is wrong, subsequent instructions and their results are purged. Instructions ahead of the predicted branch proceed normally, instruction fetching resumes along the correct path, and the history bits are revised.

The number of speculative branches that have not yet been allocated (and are predicted as not taken) is limited only by the space available in the pipeline (the branch execute unit, the BIQ, and the IQ). The number of speculative branches that hit in the BTB is limited by the fact that a fetch address group can contain only one branch that is allocated in the BTB. The e500 continues to predict branch targets until the IQ is full or until a branch resolves as mispredicted.

Instructions after any unresolved branches execute speculatively, but they cannot complete until all older branches are resolved.

Because the CQ ensures in-order completion, instructions cannot update architecture-defined registers (in write-back stage) until all previous instructions have completed. So speculatively executed instructions complete only after the predicting branch instruction, for which correctly resolving the prediction is a completion requirement.

In case of a misprediction, the e500 easily redirects its machine state because the architectural state is not updated. Any instructions dispatched after a mispredicted branch instruction are flushed from the CQ, and any results are flushed from the rename registers.

4.4.1.3 Branch Table Buffer (BTB)

The e500 implements a branch table buffer (BTB) consisting of 512 entries organized as 128 rows with 4-way set associativity. Entries can be loaded and locked into the BTB with the **bblels** instruction, which allows for deterministic branch latencies.

In the e500, the address in a BTB entry corresponding to a branch instruction is not necessarily the address of the branch instruction. To understand what address is stored in the entry, one must first understand the instruction fetch mechanism.

The e500 attempts to fetch as many as four instructions in a single access. The instructions that are fetched simultaneously are considered to belong to a fetch group; the address issued by the fetch unit that results in the delivery of the fetch group is called a fetch group address (FGA). The number of instructions in a fetch group depends on the FGA. A cache block in the e500 holds eight instructions. Even when the cache is disabled, instructions are fetched eight instructions at a time and placed in a hardware line fill buffer from which a fetch group is delivered to the core. (Here, the term ‘cache line’ is used to refer generically to a group of eight instructions that map to a single cache line regardless of whether the cache is enabled.) A fetch group terminates at a cache-line boundary.

As shown in Figure 4-8, if instructions in a cache line are numbered 0–7 and the fetch group address maps to the n^{th} instruction, where $n = 0, 1, 2, 3, 4$ or 5 , instructions $n, n+1, n+2, n+3$ are in the fetch group. If $n > 5$, instructions n through 7 are part of the fetch group.

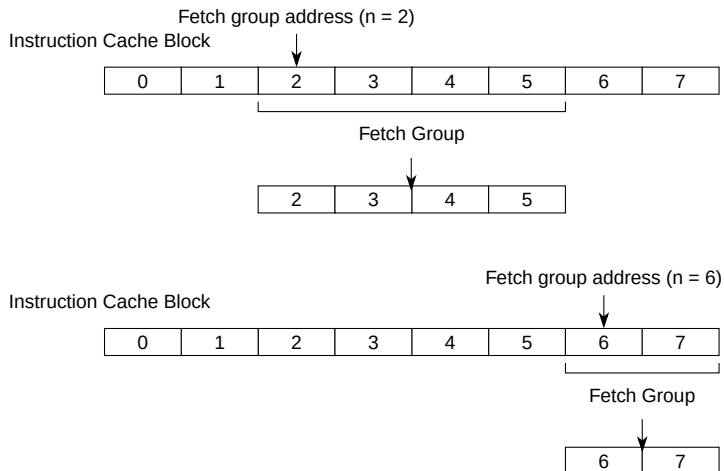


Figure 4-8. Fetch Groups and Cache Line Alignment

The BTB uses the fetch group address to predict the presence of an allocated branch rather than the address of the branch instruction. Because the same branch can be fetched at different times as a part of a different fetch group with a different fetch group address, software must lock all possible fetch group addresses that can cause the fetch of the branch instruction. The following factors affect the fetch group address of a branch instruction:

- The location of the branch instruction in the cache line
- The control flow that may allow multiple execution paths to reach the branch from different fetch group addresses.
- The presence of other branch instructions in the fetch group that precede the branch instruction under consideration

Execution Unit Timings

- Interrupts that are taken as a result of accepting an external interrupt or exceptions in instructions preceding the branch instruction in the fetch group.
- Events inside the core causing a synchronization in the pipeline during the execution of an instruction preceding the branch instruction in the fetch group
- The presence of instructions such as **isync** before the branch instruction in the fetch group

Figure 4-9 shows all possible fetch group addresses (FGAs) that can be associated with a branch instruction. In this flow chart, the location of an instruction is i if it is the i^{th} instruction ($i=0..7$) from the beginning of a cache line. The address of an instruction a_i refers to the address of the i^{th} instruction in the cache line ($i=0..7$). The condition IB refers to the case where either a synchronizing instruction (such as **isync**) or a branch instruction whose prediction is locked into the BTB occurs at some location. The branch instruction under consideration is identified as b .

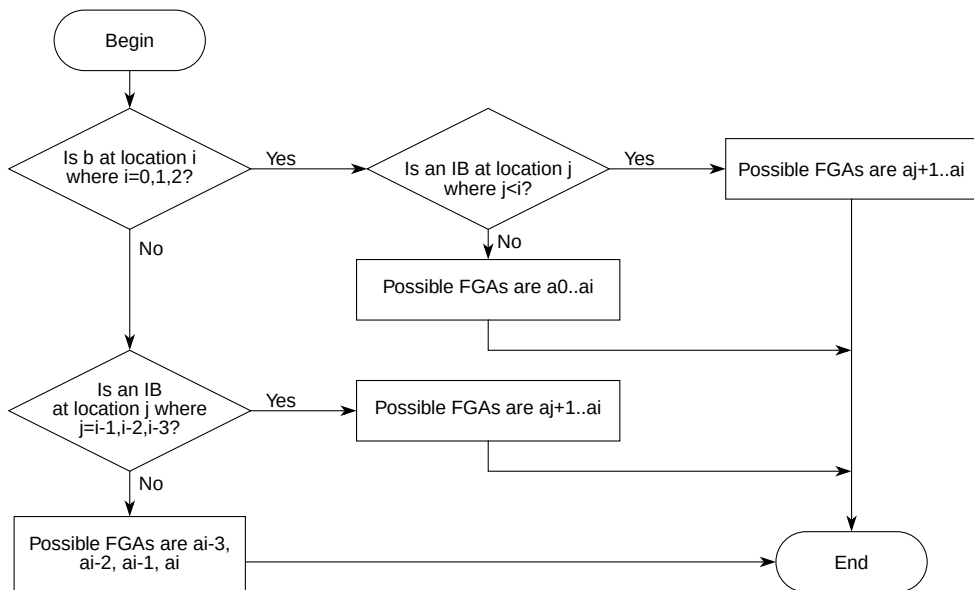


Figure 4-9. Fetch Group Addresses

Note that branch instructions that are not allocated into the BTB (either because they have never been taken or because they have been cast out of the BTB) can fall in the same fetch group. For example, the following code sequence has two branch instructions that fall into the same fetch group the first time the sequence is executed:

```

A:    add
      b1
      add
      b2
  
```

Assuming that fetching begins at A and that the sequence lies within a single cache block, all four instructions are included in the same fetch group, including both branches because they have not been taken and therefore do not have BTB entries.

At execution, b1 is not-taken, but b2 mispredicts and resolves as taken in the execute stage, at which point, the branch instruction prediction (strongly-taken) is allocated for b2 at A.

Later, b1 is taken and thus mispredicted. The BTB entry for address A becomes allocated for b1, replacing the prediction for b2. If we fetch b2 again, it is now a BTB miss and the default prediction is used. However, if the default prediction is incorrect, a separate BTB entry is allocated for b2 (at address A+8).

So now that both branch instructions are allocated in the BTB, they can no longer be in the same fetch group.

Note that rather than allowing branch predictions to change dynamically, the programmer can explicitly lock the predictions into the BTB.

The typical sequence of instructions to lock a branch address into a BTB entry is as follows:

```
mtspr BBEAR, rA
mtspr BBтар, rB
bblsls
```

The typical sequence of instructions to clear locked entries individually is as follows:

```
mtspr BBEAR, rA
bbelr
```

To guarantee atomicity, these instruction sequences should be protected by **lwarx** and **stwcx** instructions.

4.4.1.3.1 BTB Locking APU Programming Model

The BTB APU programming model includes the following register resources:

- The following BTB locking APU registers.
 - Branch buffer entry address register (BBEAR)
 - Branch buffer target address register (BBтар)
 - Branch unit control and status register (BUCSR)

These registers are described in Section 2.9, “Branch Target Buffer (BTB) Registers.”

- MSR fields. The user branch locking enable bit (UBLE) is defined in the MSR. Setting MSR[UBLE] allows user mode programs to lock or unlock BTB entries. See Section 2.5.1, “Machine State Register (MSR).”

The BTB also defines the following instructions, described in Section 3.11.2, “Branch Target Buffer (BTB) Locking Instructions”:

- Branch Buffer Load Entry and Lock Set (**bblels**)
- Branch Buffer Entry Lock Reset (**bblelr**)

4.4.1.3.2 BTB Operations

This following BTB operations are controlled through the BUCSR:

- BTB disabling. BUCSR[BPEN] is used to enable or disable the BTB. The BTB is enabled when the bit is set and disabled when it is cleared. When it is disabled, BTB contents are not used to predict the branch targets and the BTB is not updated as a result of executing branch, **bblels**, or **bblelr** instructions. However, when it is disabled, the BTB maintains its contents and any locks, which can be used again when the BTB is reenabled.
- BTB overlocking. BUCSR[BBLO] is used to report an overlocking status to the program. It is a sticky bit and once set, remains set until explicitly cleared by writing a 0 to it with an **mtspr** instruction.
- BTB unable to lock. If **bblels** cannot set the BTB lock, BUCSR[BBUL] is set. It is a sticky bit.
- BTB invalidation. Flash invalidation of the BTB is accomplished by writing BUCSR[BBFI] with a 0 and then a 1 using **mtspr** instructions.
- BTB lock clearing. BUCSR[BBLFC] is used to perform a flash lock clear (unlocking) of all locked BTB entries. Writing BUCSR[BBLFC] with a 0 and then a 1 flash lock clears all locked BTB entries.

4.4.1.3.3 BTB Special Cases—Phantom Branches and Multiple Matches

This following special cases can occur:

- Phantom branches. BTB entries hold effective addresses associated with a branch instruction. A process context switch might bring in another task whose MMU translations are such that it uses the same effective address for another instruction for which the BTB has an entry for a previously encountered branch. This causes the fetch unit to redirect instruction fetch to the BTB's target address. Later, during execution of the instruction, the hardware realizes the error and evicts the BTB entry. However, locked BTB entries are not evicted. Hardware guarantees correct execution under locked phantom branches, but performance may suffer.
- Multiple matches. Multiple matches occur if an address stored in multiple entries in different ways of the set-associative BTB match the same fetch address. The e500 hardware prevents this by ensuring that an entry is unique when it is stored. When an entry is being written as a result of a branch or through **bblels**, if hardware encounters a preexisting entry with the same fetch address, the entry is overwritten if it is not locked or the BTB is not updated if the entry has already been locked.

4.4.2 Load/Store Unit Queueing Structures

The core complex provides a 32-byte interface (the width of a cache block) to efficiently move data between the GPRs and the data cache. The LSU provides the logic required to calculate effective addresses, handles data alignment to and from the data cache, provides sequencing for load/store multiple operations, and interfaces with the core interface unit. Write operations to the data cache can be performed on a byte, half-word, word, or double-word basis.

This section describes the LSU queues that support the L1 data cache. See Section 12.3.5, “Load/Store Operations,” for more information on architectural coherency implications of load/store operations and the LSU on the core complex. Also, see Section 4.4.5, “Load/Store Unit Execution Timing,” for more information on other aspects of the LSU and instruction scheduling considerations.

The instruction and data caches are integrated with the LSU, instruction unit, and core interface unit in the memory subsystem of the core complex as shown in Figure 4-10.

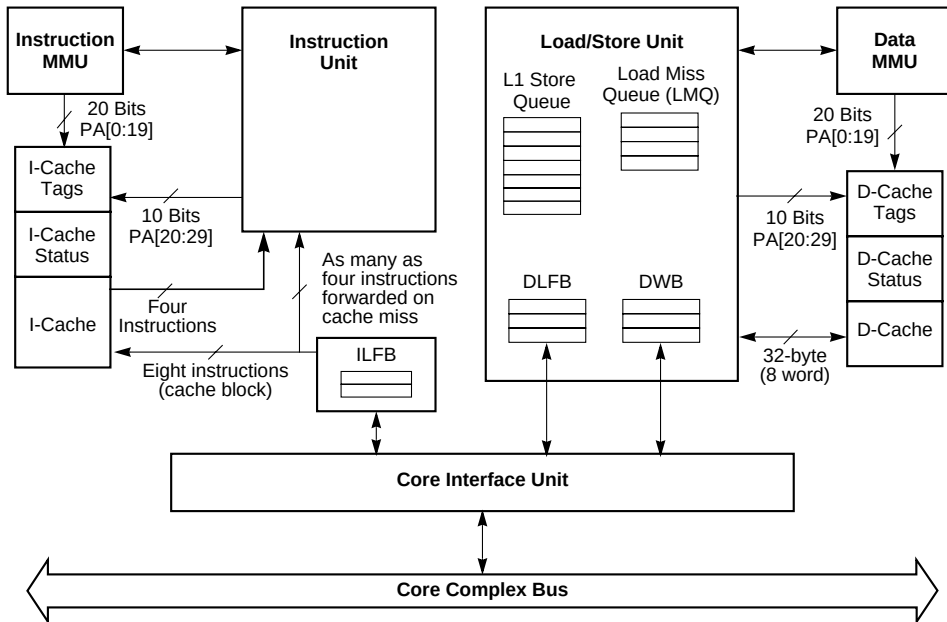


Figure 4-10. Cache/Core Interface Unit Integration

The following sections briefly describe the LSU, the instructions unit, the core interface unit and the core complex bus.

When free of data dependencies, cacheable loads execute in the LSU in a speculative manner with a maximum throughput of one per cycle and a total 3-cycle latency for integer

loads. Data returned from the cache on a load is held in a rename buffer until the completion logic commits the value to the processor state.

Table 4-2. Load and Store Queues

Queue	Description
LSU store queue	Stores cannot execute speculatively and are held in the seven-entry store queue, shown in Figure 4-10, until completion logic indicates that the store instruction is to be committed. The store queue arbitrates for L1 data cache access. When arbitration succeeds, data is written to the data cache and the store is removed from the store queue. If a store is caching-inhibited, the operation moves through the store queue to the rest of the memory subsystem.
LSU L1 load miss queue (LMQ)	As loads reach the LSU, it tries to access the cache. On a hit, the cache returns the data. If there is a miss, the LSU allocates an LMQ entry and a DLFB entry. The LSU then queues a bus transaction to read the line. If a subsequent load hits, the cache returns the results. If a subsequent load misses, the LSU allocates a second LMQ entry and, if the load is to a different cache line than the outstanding miss, it allocates the second DLFB entry and queues a second read transaction on the bus. If the load miss is to the same cache line as the outstanding miss, the LSU need not allocate the second DLFB entry. The LSU continues processing load hits and load misses until one of the following conditions occurs: • The four-entry LMQ is full and there is a fifth load miss. • The LSU tries to perform a load miss, all three of the line fill entries in the DLFB are full, and the load is not to any of the cache lines that are represented in the DLFB.
LSU data line fill buffer (DLFB)	The three DLFB entries are used for loads and cacheable stores. Stores are allocated in the DLFB so loads can access data from the store immediately (loads cannot access data from the L1 store queue). Also, by using the DLFB entries for stores, the LSU frees L1 store queue entries. Multiple cacheable store misses are merged in DLFB.
LSU data write buffer (DWB)	When a full line of data is available in the DLFB, the data cache is updated. If a data cache update requires a cache line to be evicted, the line is cast out and placed in the DWB until the data has been transferred through the core interface unit to the core complex bus. If global memory's coherency needs to be maintained as a result of bus snooping, the L1 cache can also evict a line to the DWB (this is a snoop push). Cast-out and snoop push writes from the L1 cache are cache-line aligned (critical word is not written first), regardless of which word in a modified cache line is accessed. One DWB entry is dedicated for snoop pushes, one is for cast outs, and one can be used for either.

The core interface unit handles all bus transactions initiated by the ILFB, DLFB, and DWB. The core interface unit handles all ordering and bus protocol and is the interface between the core complex and the external memory and caches.

The core interface unit performs transactions through the core complex bus by transferring either the critical double word first (8 bytes) or the critical quad word first (16 bytes). It then forwards the transaction to the instruction or data line fill buffer critical double word first. The core complex bus also captures snoop addresses for the L1 data cache and the memory reservation (**lwarx** and **stwcx**.) operations.

If a load/store miss aliases to the same line as a previously outstanding miss, the LSU halts new access until this stall condition is resolved.

The following sections briefly describe the load/store unit, the instructions unit, the core interface unit and the core complex bus.

4.4.2.1 Load/Store Unit (LSU)

The data cache supplies data to the general-purpose registers (GPRs) by means of the load/store unit (LSU). The core complex LSU is directly coupled to the data cache with a 32-byte interface (the width of a cache block) to allow efficient movement of data to and from the GPRs. The LSU provides all of the logic required to calculate effective addresses, handles data alignment to and from the data cache, provides sequencing for load/store multiple operations, and interfaces with the core interface unit. Write operations to the data cache can be performed on a byte, half-word, word, or double-word basis.

When free of data dependencies, cacheable loads execute in the LSU in a speculative manner with a maximum throughput of one per cycle and a total three-cycle latency for integer loads. Data returned from the cache on a load is held in a rename buffer until the completion logic commits the value to the processor state.

This section describes the LSU queues that support the L1 data cache. See Section 12.3.5, “Load/Store Operations,” for more information on architectural coherency implications of load/store operations and the LSU on the core complex. Also, see Section 4.4.5, “Load/Store Unit Execution Timing,” for more information on other aspects of the LSU and instruction scheduling considerations.

4.4.3 Simple and Multiple Unit Execution Timings

The e500 has two single-cycle units (SU1, SU2) and one multiple-cycle unit (MU). The SUs execute all Book E logical and computational instructions except multiplies and divides and SPE APU logical, shift, and splat instructions. The MU executes multiplies, divides, and SPE APU arithmetic instructions. Divide latency depends upon the operand data and ranges from 4 to 35 cycles. While the divide is executing, the rest of the MU is available for additional instructions (non-blocking divide). When either **rA** or **rB** is zero, the divide takes 4 cycles. When the **rA** operand is representable as a 16-bit value within the instruction context (signed or unsigned), the divide takes 19 cycles. When the **rA** operand can be represented as an 8-bit value within context, the divide takes 11 cycles. All other cases take 35 cycles. Table 4-6 lists SU and MU execution latencies. As Table 4-6 shows, most instructions executed in the SU have a single-cycle execution latency.

4.4.4 SPE Floating-Point Execution Timing

The MU executes SPE single-precision floating-point arithmetic operations, which are largely compliant with the IEEE-754 floating-point standard. Software exception handling is required to achieve full IEEE 754-compliance because the IEEE floating-point exception model is not fully implemented in hardware.

Floating-point arithmetic instructions, except for divide, execute with 4-cycle latency and 1-cycle throughput. Single-precision floating-point multiply, add, and subtract instructions execute in the four-stage pipeline MU.

If **rA** or **rB** is zero, a floating-point divide takes 4 cycles. All other cases take 29 cycles.

Table 4-8 shows floating-point instruction execution timing.

4.4.5 Load/Store Unit Execution Timing

The load/store unit executes instructions that move data between the GPRs and the memory unit of the core (made up of the L1 caches and the core interface unit buffers). Figure 4-2 shows the block diagram for the load/store unit.

The execution of most load instructions is pipelined in the three LSU stages, during which the effective address is calculated, MMU translation are performed, the data cache array and tags are read, and cache way selection and data alignment are performed. Cacheable loads, when free of data dependencies, execute in a speculative manner with a maximum throughput of one instruction per cycle and 3-cycle latency. Data returned from the cache is held in a rename register until the completion logic commits the value to the processor state. The data cache array is updated after a store instruction is retired.

Stores cannot be executed speculatively and must be held in the store queue until the completion logic signals that the store instruction is to be committed.

If operands are misaligned, additional latency may be incurred either for an alignment exception or for additional bus accesses. Load instructions that miss in the cache block cause subsequent cache accesses during the cache line refill. Table 4-7 gives load and store instruction execution latencies.

4.4.5.1 Effect of Operand Placement on Performance

The location and alignment of operands in memory may affect performance of memory accesses, in some cases significantly, as shown in Table 4-4.

Alignment of memory operands on natural boundaries guarantees the best performance. For the best performance across the widest range of implementations, the programmer should assume the performance model described in Section 3.1, “Operand Conventions.”

The effect of alignment on memory operation performance is the same for big- and little-endian addressing modes except for multiple operations, which cause an alignment interrupt in little-endian mode.

In Table 4-4, optimal means that one effective address (EA) calculation occurs during the memory operation. Fair means that multiple EA calculations occur during the operation, which may cause additional bus activities with multiple bus transfers. Poor means that an alignment interrupt is generated by the memory operation.

4.5 Memory Performance Considerations

Because the e500 has a maximum instruction throughput of two instructions per clock cycle, lack of memory bandwidth can affect performance. To maximize performance, the e500 must be able to read and write data efficiently. If a system has multiple bus devices, one device may experience long memory latencies while another device (for example, a direct-memory access controller) is using the external bus.

4.6 Instruction Latency Summary

Instruction timing is shown in Table 4-3 through Table 4-7. The latency tables use the following conventions:

- Pipelined load/store and floating-point instructions are shown with cycles of total latency and throughput cycles separated by a colon.
- Floating-point instructions with a single entry in the cycles column are not pipelined. Integer divide instructions are also not pipelined with other divides.

Table 4-3 through Table 4-7 list latencies associated with instructions executed by each execution unit. Figure 4-3 describes branch instruction latencies.

Table 4-3. Branch Operation Execution Latencies

Mnemonic	Unit	Cycles
b[l][a]	BU	1
bc[l][a]	BU	1
bcctr[l]	BU	1
bclr[l]	BU	1

Table 4-4 lists system operation instruction latencies.

Table 4-4. System Operation Instruction Execution Latencies

Mnemonic	Unit	Cycles	Serialization
isync	— ¹	0	Refetch
mfmsr	SU1	4	—
mfspr (LR or CTR)	SU1 or SU2	1	—
mfspir (XER)	SU1	4	Move-from
mfspir (other)	SU1	4	—
mtmsr	SU1	2	Move-to
mtspr (LR or CTR)	SU1 or SU2	1	Move-to
mtspr (XER)	SU1	2	Move-to, post-sync
mtspr (other)	SU1	2	Move-to

Table 4-4. System Operation Instruction Execution Latencies (continued)

Mnemonic	Unit	Cycles	Serialization
msync			—
rfi	— ¹	0	Refetch
rfci	— ¹	0	Refetch
rfmci	— ¹	0	Refetch
sc	— ¹	0	Refetch
tlbsync	LSU	3:18	Store
wrttee	SU1	1	Post-sync, move-to
wrtteei	SU1	1	Post-sync, move-to

¹ Refetch serialized instructions (if marked with a 0-cycle execution time) do not have an execute stage, and all refetch serialized instructions have 1 cycle between the time they are completed and the time the target/sequential instruction enters the fetch1 stage.

Table 4-5 lists condition register logical instruction latencies.

Table 4-5. Condition Register Logical Execution Latencies

Mnemonic	Unit	Cycles	Serialization
crand	BU	1	—
crandc	BU	1	—
creqv	BU	1	—
crnand	BU	1	—
crnor	BU	1	—
cror	BU	1	—
crorc	BU	1	—
crxor	BU	1	—
mcrf	BU	1	—
mcrxr	BU	1	Pre-sync, post-sync
mfcrr	SU1	1	Move-from
mtcrf (single field) ¹	SU1	1	—
mtcrf (multiple fields)	BU	2	Move-to, pre-sync, post-sync

¹ **mtcrf** of a single field is executed by SU1 in a single cycle and is not serialized.

Table 4-6 lists integer instruction latencies.

Table 4-6. SU and MU PowerPC Instruction Execution Latencies

Mnemonic	Unit	Cycles
addc[o][.]	SU1 or SU2	1 ³
adde[o][.]	SU1 or SU2	1 ³
addi	SU1 or SU2	1
addic	SU1 or SU2	1
addic.	SU1 or SU2	1 ³
addis	SU1 or SU2	1
addme[o][.]	SU1 or SU2	1 ³
addze[o][.]	SU1 or SU2	1 ³
add[o][.]	SU1 or SU2	1 ³
andc[.]	SU1 or SU2	1 ³
andi.	SU1 or SU2	1 ³
andis.	SU1 or SU2	1 ³
and[.]	SU1 or SU2	1 ³
cmp	SU1 or SU2	1
cmpi	SU1 or SU2	1
cmpl	SU1 or SU2	1
cmpli	SU1 or SU2	1
cntlzw[.]	SU1	1 ³
divwu[o][.] ¹	MU	4, 11, 19, 35 ³
divw[o][.] ²	MU	4, 11, 19, 35 ³
eqv[.]	SU1 or SU2	1 ³
extsb[.]	SU1 or SU2	1 ³
extsh[.]	SU1 or SU2	1 ³
mulhwu[.]	MU	4:1 ³
mulhw[.]	MU	4:1 ³
mulli	MU	4:1
mullw[o][.]	MU	4:1 ³
nand[.]	SU1 or SU2	1 ³
neg[o][.]	SU1 or SU2	1 ³
nor[.]	SU1 or SU2	1 ³
orc[.]	SU1 or SU2	1 ³
ori	SU1 or SU2	1
oris	SU1 or SU2	1

Table 4-6. SU and MU PowerPC Instruction Execution Latencies (continued)

Mnemonic	Unit	Cycles
or[.]	SU1 or SU2	1 ³
rlwimi[.]	SU1 or SU2	1 ³
rlwinm[.]	SU1 or SU2	1 ³
rlwnm[.]	SU1 or SU2	1 ³
slw[.]	SU1 or SU2	1 ³
srawi[.]	SU1 or SU2	1 ³
sraw[.]	SU1 or SU2	1 ³
srw[.]	SU1 or SU2	1 ³
subfc[o][.]	SU1 or SU2	1 ³
subfe[o][.]	SU1 or SU2	1 ³
subfic	SU1 or SU2	1
subfme[o][.]	SU1 or SU2	1 ³
subfze[o][.]	SU1 or SU2	1 ³
subf[o][.]	SU1 or SU2	1 ³
tw	SU1 or SU2	1
twi	SU1 or SU2	1
xori	SU1 or SU2	1
xoris	SU1 or SU2	1
xor[.]	SU1 or SU2	1 ³

¹ For division by zero, latency is 4 cycles.

² For division by zero or 0x8000_0000/ 0xFFFF_FFFF, latency is 4 cycles

³ If the record bit is set, CR results are not available until after one more cycle.
A subsequent instruction can execute while CR results are generated.

Table 4-7 shows load and store instruction latencies. Load/store multiple instruction cycles are represented as a fixed number of cycles plus a variable number of cycles, where *n* represents the number of words accessed by the instruction. Pipelined load/store instructions are shown with total latency and throughput separated by a colon.

Table 4-7. Load/Store Unit (LSU) Instruction Latencies

Mnemonic	Cycles ¹	Serialization
dcba	3:1	Store
dcbf	3:1	Store
dcbst	3:1	Store
dcbt	3:1	—
dcbtst	3:1	—

Table 4-7. Load/Store Unit (LSU) Instruction Latencies (continued)

Mnemonic	Cycles ¹	Serialization
dcbz	3:1	Store
evldd	3:1	
evlddx	3:1	
evldh	3:1	
evldhx	3:1	
evldw	3:1	
evldwx	3:1	
evlhhesplat	3:1	
evlhhesplatx	3:1	
evlhossplat	3:1	
evlhossplatx	3:1	
evlhousplat	3:1	
evlhousplatx	3:1	
evlwhe	3:1	
evlwhex	3:1	
evlwhos	3:1	
evlwhosx	3:1	
evlwhou	3:1	
evlwhoux	3:1	
evlwhsplat	3:1	
evlwhsplatx	3:1	
evlwwsplat	3:1	
evlwwsplatx	3:1	
evstd	3:1	
evstdx	3:1	
evstdh	3:1	
evstdhx	3:1	
evstdw	3:1	
evstdwx	3:1	
evstwhe	3:1	
evstwhex	3:1	
evstwho	3:1	
evstwhox	3:1	

Table 4-7. Load/Store Unit (LSU) Instruction Latencies (continued)

Mnemonic	Cycles ¹	Serialization
evstww	3:1	
evstwwex	3:1	
evstww	3:1	
evstwwox	3:1	
icbi	3:1	Store
lbz	3:1	—
lbzu	3:1 ²	—
lbzux	3:1 ²	—
lbzx	3:1	—
lha	3:1	—
lhau	3:1 ²	—
lhaux	3:1 ²	—
lhax	3:1	—
lhbrx	3:1	—
lhz	3:1	—
lhzu	3:1 ²	—
lhzux	3:1 ²	—
lhzx	3:1	—
lmw	2 + n	—
lwarx	3	Presync
lwbrx	3:1	—
lwz	3:1	—
lwzu	3:1 ²	—
lwzux	3:1 ²	—
lwzx	3:1	—
stb	3:1	Store
stbu	3:1 ²	Store
stbux	3:1 ²	Store
stbx	3:1	Store
sth	3:1	Store
sthbrx	3:1	Store
sth	3:1 ²	Store
sthux	3:1 ²	Store

Table 4-7. Load/Store Unit (LSU) Instruction Latencies (continued)

Mnemonic	Cycles ¹	Serialization
sthx	3:1	Store
stmw	3 + n	Store
stw	3:1	Store
stwbrx	3:1	Store
stwcx.	3:1	Store, presync, postsync
stwu	3:1 ²	Store
stwux	3:1 ²	Store
stwx	3:1	Store
tlbivax	3:1	—
tlbre	3:1	Pre-sync, post-sync, move-to
tlbsx	3:1	Pre-sync, post-sync, move-to
tlbwe	3:1	Pre-sync, post-sync, move-to

¹ For cache operations, the first number indicates the latency for finishing a single instruction and the second number indicates the throughput for a large number of back-to-back cache operations. The throughput cycle may be larger than the initial latency because more cycles may be needed for the data to reach the cache. If the cache remains busy, subsequent cache operations cannot execute.

² Load and store update instructions are broken into two instructions at dispatch, a load or store instruction that executes in the LSU and an **addi** that executes in either SU. See Section 4.3.3.1, "GPR and CR Rename Register Operation."

Table 4-8 lists instruction latencies for SPE APU computational and logical instructions. SPE loads and stores are executed by the LSU and are described in Table 4-7.

Table 4-8. Signal Processing SPE APU Instruction Latencies

Mnemonic	Unit	Cycles
brinc	SU1 or SU2	1
efsabs	SU1 or SU2	1
efsadd	MU	4:1
efscfsf	MU	4:1
efscfsi	MU	4:1
efscfuf	MU	4:1
efscfui	MU	4:1
efscmpeq	SU1	1
efscmpgt	SU1	1
efscmplt	SU1	1
efscstsf	MU	4:1
efscstsi	MU	4:1

Table 4-8. Signal Processing SPE APU Instruction Latencies (continued)

Mnemonic	Unit	Cycles
efsctsiz	MU	4:1
efscuf	MU	4:1
efstui	MU	4:1
efstuiiz	MU	4:1
efsdv	MU	4, 29
evsel	SU1	1
efsmul	MU	4:1
efsnabs	SU1 or SU2	1
efsneg	SU1 or SU2	1
efssub	MU	4:1
efststeg	SU1 or SU2	1
efststgt	SU1 or SU2	1
efststlt	SU1 or SU2	1
evabsw	SU1	1
evaddiw	SU1	1
evaddsmiaaw	MU	4:1
evaddssiaaw	MU	4:1
evaddumiaaw	MU	4:1
evaddusiaaw	MU	4:1
evaddw	SU1	1
evand	SU1	1
evandc	SU1	1
evcmpeq	SU1	1
evcmpgts	SU1	1
evcmpgtu	SU1	1
evcmplt	SU1	1
evcmpltu	SU1	1
evcntlsw	SU1	1
evcntlzw	SU1	1
evdivws	MU	4, 11, 19, 35
evdivwu	MU	4, 11, 19, 35
eveqv	SU1	1
evextsb	SU1	1

Table 4-8. Signal Processing SPE APU Instruction Latencies (continued)

Mnemonic	Unit	Cycles
evextsh	SU1	1
evfsabs	SU1	1
evfsadd	MU	4:1
evfscfsf	MU	4:1
evfscfsi	MU	4:1
evfscfuf	MU	4:1
evfscfui	MU	4:1
evfscmpeq	MU	4:1
evfscmpgt	MU	4:1
evfscmplt	MU	4:1
evfsctsf	MU	4:1
evfsctsi	MU	4:1
evfsctsiz	MU	4:1
evfsctuf	MU	4:1
evfsctui	MU	4:1
evfsctuiz	MU	4:1
evfsdiv	MU	4, 29
evfsmul	MU	4:1
evfsnabs	SU1	1
evfsneg	SU1	1
evfssub	MU	4:1
evfststeq	SU1	1
evfststgt	SU1	1
evfststlt	SU1	1
evmergehi	SU1	1
evmergehilo	SU1	1
evmergelo	SU1	1
evmergelohi	SU1	1
evmhegsmfaa	MU	4:1
evmhegsmfan	MU	4:1
evmhegsmiaa	MU	4:1
evmhegsmian	MU	4:1
evmhegumiaa	MU	4:1

Table 4-8. Signal Processing SPE APU Instruction Latencies (continued)

Mnemonic	Unit	Cycles
evmhegumian	MU	4:1
evmhesmf	MU	4:1
evmhesmfa	MU	4:1
evmhesmfaaw	MU	4:1
evmhesmfanw	MU	4:1
evmhesmi	MU	4:1
evmhesmia	MU	4:1
evmhesmiaaw	MU	4:1
evmhesmianw	MU	4:1
evmhessf	MU	4:1
evmhessfa	MU	4:1
evmhessfaaw	MU	4:1
evmhessfanw	MU	4:1
evmhessiaaw	MU	4:1
evmhessianw	MU	4:1
evmheumi	MU	4:1
evmheumia	MU	4:1
evmheumiaaw	MU	4:1
evmheumianw	MU	4:1
evmheusiaaw	MU	4:1
evmheusianw	MU	4:1
evmhogsmfaa	MU	4:1
evmhogsmfan	MU	4:1
evmhogsmiaa	MU	4:1
evmhogsmian	MU	4:1
evmhogumiaa	MU	4:1
evmhogumian	MU	4:1
evmhosmf	MU	4:1
evmhosmfa	MU	4:1
evmhosmfaaw	MU	4:1
evmhosmfanw	MU	4:1
evmhosmi	MU	4:1
evmhosmia	MU	4:1

Table 4-8. Signal Processing SPE APU Instruction Latencies (continued)

Mnemonic	Unit	Cycles
evmhosmiaaw	MU	4:1
evmhosmianw	MU	4:1
evmhossf	MU	4:1
evmhossfa	MU	4:1
evmhossfaaw	MU	4:1
evmhossfanw	MU	4:1
evmhossiaaw	MU	4:1
evmhossianw	MU	4:1
evmhoumi	MU	4:1
evmhoumia	MU	4:1
evmhoumiaaw	MU	4:1
evmhoumianw	MU	4:1
evmhousiaaw	MU	4:1
evmhousianw	MU	4:1
evmra	MU	4:1
evmwshmf	MU	4:1
evmwshmfafa	MU	4:1
evmwshsmi	MU	4:1
evmwshsmia	MU	4:1
evmwshssf	MU	4:1
evmwshssfa	MU	4:1
evmwshumi	MU	4:1
evmwshumia	MU	4:1
evmwslmf	MU	4:1
evmwslmfafa	MU	4:1
evmwslmfafaaw	MU	4:1
evmwslmfafanw	MU	4:1
evmwslsmiaaw	MU	4:1
evmwslsmianw	MU	4:1
evmwslssf	MU	4:1
evmwslssfa	MU	4:1
evmwslssafaaw	MU	4:1
evmwslssfafanw	MU	4:1

Table 4-8. Signal Processing SPE APU Instruction Latencies (continued)

Mnemonic	Unit	Cycles
evmwlsiaaw	MU	4:1
evmwlsianw	MU	4:1
evmwлуми	MU	4:1
evmwлумia	MU	4:1
evmwлумiaaw	MU	4:1
evmwлумianw	MU	4:1
evmwлusiaaw	MU	4:1
evmwлusianw	MU	4:1
evmwsmf	MU	4:1
evmwsmfa	MU	4:1
evmwsmfaa	MU	4:1
evmwsmfan	MU	4:1
evmwsmi	MU	4:1
evmwsmia	MU	4:1
evmwsmiaa	MU	4:1
evmwsmian	MU	4:1
evmwssf	MU	4:1
evmwssfa	MU	4:1
evmwssfaa	MU	4:1
evmwssfan	MU	4:1
evmwumi	MU	4:1
evmwumia	MU	4:1
evmwumiaa	MU	4:1
evmwumian	MU	4:1
evnand	SU1	1
evneg	SU1	1
evnor	SU1	1
evor	SU1	1
evorc	SU1	1
evrlw	SU1	1
evrlwi	SU1	1
evrndw	SU1	1
evslw	SU1	1

Table 4-8. Signal Processing SPE APU Instruction Latencies (continued)

Mnemonic	Unit	Cycles
evslwi	SU1	1
evsplatfi	SU1	1
evsplatfi	SU1	1
evsrwis	SU1	1
evsrwiu	SU1	1
evsrws	SU1	1
evsrwu	SU1	1
evsubfsmiaaw	MU	4:1
evsubfssiaaw	MU	4:1
evsubfumiaaw	MU	4:1
evsubfusiaaw	MU	4:1
evsubfw	SU1	1
evsubifw	SU1	1
evxor	SU1	1

4.7 Instruction Scheduling Guidelines

This section provides an overview of instruction scheduling guidelines, followed by detailed examples showing how to optimize scheduling with respect to various pipeline stages. Performance can be improved by avoiding resource conflicts and scheduling instructions to take fullest advantage of the parallel execution units. Instruction scheduling can be improved by observing the following guidelines:

- To reduce branch mispredictions, separate the instruction that sets CR bits from the branch instruction that evaluates them. Because there can be no more than 26 instructions in the processor (with the instruction that sets CR in CQ0 and the dependent branch instruction in IQ11), there is no advantage to having more than 24 instructions between them.
- When branching to a location specified by the CTR or LR, separate the **mtspr** instruction that initializes the CTR or LR from the dependent branch instruction. This ensures the register values are immediately available to the branch instruction.
- Schedule instructions so two can be dispatched at a time.
- Schedule instructions to minimize stalls due to busy execution units.
- Avoid scheduling high-latency instructions close together. Interspersing single-cycle latency instructions between longer-latency instructions minimizes the effect that instructions such as integer divide can have on throughput.
- Avoid using serializing instructions.

- Schedule instructions to avoid dispatch stalls. As many as 14 instructions can be assigned CR and GPR renames and can be assigned CQ entries; therefore, 14 instructions can be in the execute stages at any one time. (However, note the exception of load or store with update instructions, which are broken into two instructions at dispatch.)
- Avoid branches where possible; favor not-taken branches over taken branches.

The following sections give detailed information regarding optimizing code for the e500 pipeline stages.

4.7.1 Fetch/Branch Considerations

The following is a list of branch instructions and the resources required to avoid stalling the fetch unit in the course of branch resolution:

- The **bclr** instruction requires LR availability for resolution.
- The branch conditional on counter decrement and the CR condition requires CTR availability or the CR condition must be false.

4.7.1.1 Dynamic Prediction versus No Branch Prediction

No branch prediction ($\text{BUCSR}[\text{BPEN}] = 0$) means that the e500 predicts every branch as not taken. The dynamic predictor is ignored. Sometimes this simplistic prediction is superior, either through informed guessing or through available profile-directed feedback. Run-time for code with no prediction is more nearly deterministic, which can be useful in embedded systems.

Note that disabling an enabling the BTB (by clearing and setting BPEN) do not affect the BTB's contents or locks.

4.7.1.1.1 Position-Independent Code Example

Position-independent code is used when not all addresses are known at compile time or link time. Because performance is typically not good, position-independent code should be avoided when possible.

4.7.2 Dispatch Unit Resource Requirements

The following is a list of resources required to avoid stalls in the dispatch unit (IQ0–IQ2 are the two dispatch entries in the instruction queue):

- Appropriate issue queue is available.
- The CQ is not full.
- Previous instructions to the IQ dispatch entries must dispatch.

The following sections describe how to optimize code for dispatch.

4.7.2.1 Dispatch Groupings

Maximum dispatch throughput is two instructions per cycle. The dispatch process includes a CQ available check, an issue queue available check, and a branch ready check.

The dispatcher can send two instructions to the two issues queues, with a maximum of two to the GIQ and one to the BIQ. Only one MU instruction can be dispatched per cycle, so the sequence **evaddw**, **evldd**, **evmwsmafaa** requires three cycles to dispatch. Only one load or store instruction can dispatch per cycle.

The dispatcher can rename as many as two GPRs per cycle, so a two-instruction dispatch window composed of **add** and **mulli** could be dispatched in one cycle.

Note that a load/store update form (for example, **lwzu**), requires a rename register for the update. This means an **lwzu** needs two renames. The restriction to two GPR renames in a dispatch group means that the sequence, **lwzu**, **add**, cannot be dispatched in one cycle.

4.7.3 Issue Queue Resource Requirements

Instructions cannot be issued unless the specified execution unit is available. The following sections describe how to optimize use of the three issue queues.

4.7.3.1 General Issue Queue (GIQ)

As many as two instructions can be dispatched to the four-entry general issue queue (GIQ) per cycle. As many as two instructions can be issued in any order from GIQ0 and GUIQ1. LSU, MU, SU1, and SU2 reservation stations from the bottom two GIQ entries.

Issuing instructions out-of-order can help in a number of situations. For example, if the MU is busy and a multiply is stalled at the bottom GIQ entry, the instruction in the next GIQ entry can be issued to LSU or SU1, bypassing that multiply.

4.7.4 Completion Unit Resource Requirements

The e500 completion queue has 14 entries, so as many as 14 instructions can be in the execution window. The following resources are required to avoid stalls in the completion unit; note that the two completion entries are described as CQ0–CQ1, where CQ0 is located at the end of the CQ (see Figure 6-4). Requirements for retiring instructions from CQ0–CQ1 are as follows:

- Instruction must have finished execution.
- Previous instructions that must be retired by the CQ retirement entries.
- Instruction in CQ1 must not cause an exception.

- Instruction in CQ1 must be integer (SU1 only), floating-point, load, **dcbt**, data streaming, resolved branches.

4.7.4.1 Completion Groupings

The e500 can retire as many as two instructions per cycle. Only two renames of a given type can be retired per cycle.

4.7.5 Serialization Effects

The e500 supports refetch, execution, and store serialization.

Refetch serialized instructions include **isync**, **rfi**, **sc**, **mtspr[XER]**, and any instruction that creates a rising edge on XER[SO]. Refetch serialization forces a pipeline flush when the instruction is the oldest in the machine. Avoid these instructions in performance critical code.

Note that XER[SO] is a sticky bit for XER[OV] updates, so avoiding toggling XER[SO] often means avoiding these instructions (overflow-record, O form).

Execution-serialized instructions wait until the instruction is the oldest in the machine to begin executing. Tables in Section 4.6, “Instruction Latency Summary,” list execution-serialized instructions, which include **mtspr**, **mfspir**, CR logical instructions, and carry consuming instructions (such as **adde**).

4.7.6 Execution Unit Considerations

The following sections describes how to optimize use of the execution units.

4.7.6.1 SU Considerations

Each SU has one reservation station in which instructions are held until operands are available.

Figure 4-1 shows a case where one SU is blocked and the second makes progress.

Also note that some SU1 instructions take more than one cycle and that some are not fully pipelined. The most common 2-cycle instructions are **mtspr** and **mfspir**.

The following instructions are not fully pipelined when their record bit is set: **extsb**, **extsh**, **rlwimi**, **rlwinm**, **rlwnm**, **slw**, and **srw**. These instructions return GPR data after the first cycle but continue executing into a second cycle to generate the CR result.

4.7.6.2 MU Considerations

The MU has one reservation station. Although **mtspr[CTR]/mtspr[LR]** instructions are execution serialized, if data is available, their values are forwarded to the BU as soon as they are in the reservation station.

4.7.6.2.1 Load/Store Interaction

When loads and stores are intermixed, stores normally lose arbitration to the cache. A store that repeatedly loses arbitration can stay in the core interface unit store queue much longer than 3 cycles, which is not normally a performance problem because a store in this queue is effectively part of the architecture-defined state. However, sometimes—including if the store queue fills up or if a store causes a pipeline stall (as in a partial address alias case of store to load)—the arbiter gives higher priority to the store, guaranteeing forward progress.

4.7.6.2.2 Misalignment Effects

Misalignment, particularly the back-to-back misalignment of loads, can cause strange performance effects. The e500 splits misaligned transactions into two transactions, so misaligned load latency is at least 1 cycle greater than the default latency.

For loads that hit in the cache, the throughput of the LSU is degraded to one misaligned load every 3 cycles. Similarly, stores can be translated at a rate of one store per 3 cycles. Additionally, after translation, each misaligned store is treated as two separate store queue entries, each requiring a cache access.

A word or half-word storage access requires multiple accesses if it crosses a double-word boundary. Extended vector loads and stores cause alignment exceptions if they cross their natural alignment boundaries (as show in Figure 4-9).

Table 4-9. Natural Alignment Boundaries for Extended Vector Instructions

Instruction	Boundary
evld{d,w,h} evld{d,w,h}x evstd{d,w,h} evstd{d,w,h}x	Double-word
evlwwsplat{x} evlwhe{x} evlwhe{x} evlwhe{x} evlwhe{x} evlwhe{x} evlwhe{x} evlwhe{x} evlwhe{x}	Word
evlhhesplat{x} evlhhouplat{x} evlhhouplat{x}	Half

Frequent unaligned accesses are discouraged because of the impact on performance.

Note the following:

- Accesses that cross a translation boundary may be restarted. (that is, a misaligned access that crosses a page boundary is entirely restarted if the second portion of the access causes a TLB miss. This may result in the first portion being accessed twice.)
- Accesses that cross a translation boundary where the endianness changes cause a byte-ordering DSI exception.
- Future generations of high-performance microprocessors that implement the PowerPC architecture may experience greater misalignment penalties.

4.7.6.2.3 Load Miss Pipeline

The e500 supports as many as four outstanding load misses in the load miss queue (LMQ), shown in Figure 4-10. Table 4-10 shows a load followed by a dependent **add**. Here, the load misses in the data cache and the full line is reloaded into the data cache.

Table 4-10. Data Cache Miss, L2 Cache Hit Timing

Instruction	0	1	2	3	4	5	6
lwz r4,0x0(r9)	E0	E1	Miss	LMQ0	LMQ0/E2	C	
add r5,r4,r3	—	—	—	—	—	E	C

If a load misses in the L1 data cache critical data forwarding occurs, followed shortly by the rest of the line.

Chapter 5

Interrupts and Exceptions

This chapter provides a general description of the Book E interrupt and exception model and gives details of the additions and changes to that model that are implemented in the e500 core complex.

5.1 Overview

A note on terminology:

The Book E architecture has defined additional resources for interrupt handling; as a result, the terms ‘interrupt’ and ‘exception’ differ somewhat from their use in previous Motorola documentation, such as the *Programming Environments Manual*. Usage of these terms in this document are as follows:

- An interrupt is the action in which the processor saves its context (typically the machine state register (MSR) and next instruction address) and begins execution at a predetermined interrupt handler address with a modified MSR.
- An exception is the event that, if enabled, causes the processor to take an interrupt. Book E describes exceptions as being generated by signals from internal and external peripherals, instructions, the internal timer facility, debug events, or error conditions.

There are three categories of interrupt, described as follows:

- Non-critical interrupts—First-level interrupts that allow the processor to change program flow to handle conditions generated by external signals, errors or unusual conditions arising from program execution, or from programmable timer-related events.

These interrupts are largely identical to those previously defined by OEA portion the Power PC architecture. They use save and restore registers (SRR0/SRR1) to save state when they are taken and they use the **rfi** instruction to restore state.

Asynchronous non-critical interrupts can be masked by the external interrupt enable bit, MSR[EE].

- Critical interrupts—Critical interrupts can be taken during a non-critical interrupt or during regular program flow. They use the critical save and restore registers (CSRR0/CSRR1) to save state when they are taken and they use the **rfci** instruction

to restore state. These interrupts can be masked by the critical enable bit, MSR[CE]. Book E defines the critical input, watchdog timer, and machine check interrupts as critical interrupts, but the e500 defines a third set of resources for the machine check interrupt, as described below.

- Machine check interrupt—Similar to the Book E–defined critical interrupt type, the e500 defines a separate set of resources for the machine check interrupt. They use the machine check save and restore registers (MCSRR0/MCSRR1) to save state when they are taken and they use the **rfmci** instruction to restore state. These interrupts can be masked by the machine check enable bit, MSR[ME].

All interrupts except machine check are ordered within the two categories of noncritical and critical, such that only one interrupt of each category is reported, and when it is processed (taken), no program state is lost. Because save/restore register pairs are serially reusable, program state may be lost when an unordered interrupt is taken (see Section 5.10, “Interrupt Ordering and Masking”).

All interrupts except machine check are context synchronizing as defined in Section 3.3.3.5.2, “Context Synchronization.” A machine check interrupt acts like a context-synchronizing operation with respect to subsequent instructions; that is, a machine check interrupt need not satisfy items 1 and 2 of Section 3.3.3.5.2, “Context Synchronization,” but satisfies items 3 and 4.

5.2 e500 Interrupt Definition

This section gives an overview of additions and modifications to the Book E interrupt model as it is implemented in the e500. Specific details are also provided throughout this chapter. Except for the following, the core complex takes exceptions as specified in Book E:

- The machine check exception differs as follows:
 - It is no longer a critical interrupt but uses MCSRR0 and MCSRR1 for saving the return address and the MSR in case the machine check is recoverable.
 - Return from machine check interrupt instruction (**rfmci**) is implemented to support the return to the address saved in MCSRR0.
 - A machine check syndrome register, MCSR, is used to log the cause of the machine check (instead of ESR).

The core complex takes the machine check exception as described in Section 5.7.2, “Machine Check Interrupt.”

- The core complex supports signal processing (SPE) interrupts as specified in Section 5.7.15, “e500-Specific Exceptions.”
- The debug exception implementation does not support IAC3, IAC4, DAC3, and DAC4 comparisons.

- The core complex supports instruction address compare (IAC1 and IAC2) and data address compare (DAC1 and DAC2) for effective addresses only. Real address support is not provided.
- Floating-point and auxiliary processor interrupts are not supported.
- Data value compare (DVC) debug exceptions are not supported.
- The interrupt priorities differ from those specified in Book E as described in Section 5.11.1, “e500 Interrupt Priorities.”
- The following SPE exceptions are defined:
 - SPE unavailable exception. IVOR32 (SPR 528) contains the vector offset. See Section 5.7.15.1, “SPE APU Unavailable Exception.”
 - SPE floating-point data exception. IVOR33 (SPR 529) contains the vector offset. See Section 5.7.15.2, “SPE Floating-Point Data Exception.”
 - SPE floating-point round exception. IVOR34 (SPR 530) contains the vector offset. See Section 5.7.15.3, “SPE Floating-Point Round Exception.”
- Alignment exceptions. Vector operations can cause alignment exceptions as described in Section 5.7.6, “Alignment Interrupt.”
- The following additional bits are defined to support SPE exceptions:
 - MSR[38] is defined as the vector available bit (SPE). If this bit is clear and software attempts to execute any of the SPE instructions, the SPE unavailable exception is taken. If this bit is set, software can execute any SPE instructions.
 - ESR[24] is defined as the SPE exception bit (SPE), which is set when the processor takes an exception related to the execution of SPE instructions.
- The sources of externally generated interrupts are as defined in Book E and the machine check input signal.

5.2.1 Recoverability from Interrupts

All interrupts except machine check are recoverable. The state of the core complex (return address and MSR contents) is saved when a machine check interrupt is taken. The conditions that cause a machine check may or may not prohibit recovery.

5.3 Interrupt Registers

Table 5-1 summarizes registers used for interrupt handling.

Table 5-1. Interrupt Registers Defined by Book E

Register	Description
Book E Interrupt Registers	
Save/restore register 0 (SRR0)	On a noncritical interrupt, SRR0 is set to the current or next instruction address. When rfi is executed, instruction execution continues at the address in SRR0. In general, SRR0 contains the address of the instruction that caused the noncritical interrupt or the address of the instruction to return to after a noncritical interrupt is serviced.
Save/restore register 1 (SRR1)	When a noncritical interrupt is taken, MSR contents are placed into SRR1. When rfi is executed, the SRR1 contents are placed into the MSR. SRR1 bits that correspond to reserved MSR bits are also reserved. Note that an MSR bit that is reserved may be altered by rfi/rfci/rfmci .
Critical save/restore register 0 (CSRR0)	When a critical interrupt is taken, CSRR0 is set to the current or next instruction address. When rfci is executed, instruction execution continues at the address in CSRR0. In general, CSRR0 contains the address of the instruction that caused the critical interrupt, or the address of the instruction to return to after a critical interrupt is serviced.
Critical save/restore register 1 (CSRR1)	When a critical interrupt is taken, MSR contents are placed into CSRR1. When rfci is executed, CSRR1 contents are restored into the MSR. CSRR1 bits that correspond to reserved MSR bits are also reserved. An MSR bit that is reserved may be altered by rfi/rfci/rfmci .
Data exception address register (DEAR)	Contains the address referenced by a load, store, or cache management instruction that caused an alignment, data TLB miss, or data storage interrupt.
Interrupt vector prefix register (IVPR)	IVPR[32–47] provides the high-order 48 bits of the address of the exception processing routines. The 16-bit vector offsets are concatenated to the right of IVPR[3247] to form the address of the exception processing routine. IVPR[48–63] are reserved.
Exception syndrome register (ESR)	Provides a syndrome to differentiate between different kinds of exceptions that can generate the same interrupt type. When one of these types of interrupts is generated, bits corresponding to the specific exception that generated the interrupt are set and all other ESR bits are cleared. Other interrupt types do not affect ESR contents. The ESR does not need to be cleared by software. Table 5-2 shows ESR bit definitions. The e500 defines ESR[56] as the SPE exception bit (SPE). It is set when the processor takes an exception related to the execution of an SPE instruction. Note that the e500 implementation of the machine check interrupt uses the machine check syndrome register (MCSR) rather than the ESR.

Table 5-1. Interrupt Registers Defined by Book E (continued)

Register	Description																																														
Interrupt vector offset registers (IVORs)	Holds the quad-word index from the base address provided by the IVPR for each interrupt type. IVOR0–IVOR15 are provided for defined interrupt types. SPR numbers corresponding to IVOR16–IVOR31 are reserved. IVOR[32–47,60–63] are reserved. SPR numbers for IVOR32–IVOR63 are allocated for implementation-dependent use (IVOR32–IVOR34 (SPR 528–530) are used by the e500). IVOR assignments are shown below. <table> <tr> <th><u>IVOR Number</u></th><th><u>Interrupt Type</u></th></tr> <tr><td>IVOR0</td><td>Critical input</td></tr> <tr><td>IVOR1</td><td>Machine check</td></tr> <tr><td>IVOR2</td><td>Data storage</td></tr> <tr><td>IVOR3</td><td>Instruction storage</td></tr> <tr><td>IVOR4</td><td>External input</td></tr> <tr><td>IVOR5</td><td>Alignment</td></tr> <tr><td>IVOR6</td><td>Program</td></tr> <tr><td>IVOR7</td><td>Floating-point unavailable (not supported on the e500)</td></tr> <tr><td>IVOR8</td><td>System call</td></tr> <tr><td>IVOR9</td><td>Auxiliary processor unavailable (not supported on the e500)</td></tr> <tr><td>IVOR10</td><td>Decrementer</td></tr> <tr><td>IVOR11</td><td>Fixed-interval timer interrupt</td></tr> <tr><td>IVOR12</td><td>Watchdog timer interrupt</td></tr> <tr><td>IVOR13</td><td>Data TLB error</td></tr> <tr><td>IVOR14</td><td>Instruction TLB error</td></tr> <tr><td>IVOR15</td><td>Debug</td></tr> <tr><td>IVOR16–IVOR31</td><td>Reserved for future architectural use</td></tr> <tr><td>IVOR32</td><td>SPE APU unavailable</td></tr> <tr><td>IVOR33</td><td>SPE floating-point data</td></tr> <tr><td>IVOR34</td><td>SPE floating-point round</td></tr> <tr><td>IVOR35</td><td>Performance monitor</td></tr> <tr><td>IVOR36–IVOR63</td><td>Allocated for implementation-dependent use</td></tr> </table>	<u>IVOR Number</u>	<u>Interrupt Type</u>	IVOR0	Critical input	IVOR1	Machine check	IVOR2	Data storage	IVOR3	Instruction storage	IVOR4	External input	IVOR5	Alignment	IVOR6	Program	IVOR7	Floating-point unavailable (not supported on the e500)	IVOR8	System call	IVOR9	Auxiliary processor unavailable (not supported on the e500)	IVOR10	Decrementer	IVOR11	Fixed-interval timer interrupt	IVOR12	Watchdog timer interrupt	IVOR13	Data TLB error	IVOR14	Instruction TLB error	IVOR15	Debug	IVOR16–IVOR31	Reserved for future architectural use	IVOR32	SPE APU unavailable	IVOR33	SPE floating-point data	IVOR34	SPE floating-point round	IVOR35	Performance monitor	IVOR36–IVOR63	Allocated for implementation-dependent use
<u>IVOR Number</u>	<u>Interrupt Type</u>																																														
IVOR0	Critical input																																														
IVOR1	Machine check																																														
IVOR2	Data storage																																														
IVOR3	Instruction storage																																														
IVOR4	External input																																														
IVOR5	Alignment																																														
IVOR6	Program																																														
IVOR7	Floating-point unavailable (not supported on the e500)																																														
IVOR8	System call																																														
IVOR9	Auxiliary processor unavailable (not supported on the e500)																																														
IVOR10	Decrementer																																														
IVOR11	Fixed-interval timer interrupt																																														
IVOR12	Watchdog timer interrupt																																														
IVOR13	Data TLB error																																														
IVOR14	Instruction TLB error																																														
IVOR15	Debug																																														
IVOR16–IVOR31	Reserved for future architectural use																																														
IVOR32	SPE APU unavailable																																														
IVOR33	SPE floating-point data																																														
IVOR34	SPE floating-point round																																														
IVOR35	Performance monitor																																														
IVOR36–IVOR63	Allocated for implementation-dependent use																																														
Machine state register (MSR)	MSR[38] is defined as the vector available bit (SPE). It functions as follows: 0 If software attempts to execute an SPE instruction, an SPE APU unavailable exception is taken. 1 Software can execute any SPE instructions.																																														
e500-Specific Interrupt Registers																																															
Machine check save/restore register 0 (MCSRR0)	When a machine check interrupt is taken, MCSRR0 is set to the current or next instruction address. When rfmci is executed, instruction execution continues at the address in MCSRR0. In general, MCSRR0 contains the address of the instruction that caused the machine check interrupt, or the address of the instruction to return to after a machine check interrupt is serviced.																																														
Machine check save/restore register 1 (MCSRR1)	When a machine check interrupt is taken, MSR contents are placed into MCSRR1. When rfmci is executed, MCSRR1 contents are restored to MSR. MCSRR1 bits that correspond to reserved MSR bits are also reserved. A reserved MSR bit may be altered by rfmci.																																														
Machine check syndrome register (MCSR)	When the e500 takes a machine check interrupt, it updates MCSR to differentiate between machine check conditions. Table 5-3 lists bit e500 assignments. The MCSR also indicates whether a machine check condition is recoverable. When a condition bit is set, the core complex asserts MCP_OUT for system information. Abist status is logged in MCSR[48–54]. These read-only bits do not initiate machine check (or any other exception). An Abist bit being set indicates an error being detected in the corresponding module. Note that other Book E processors use the Book E–defined ESR for this purpose.																																														

Table 5-2 shows ESR bit definitions.

Table 5-2. Exception Syndrome Register (ESR) Definition

Bits	Name	Syndrome	Interrupt Types
32–35	—	Allocated	—
36	PIL	Illegal instruction exception	Program
37	PPR	Privileged instruction exception	Program
38	PTR	Trap exception	Program
39	FP	Floating-point operation (not supported on the e500)	Alignment, data storage, data TLB, program
40	ST	Store operation	Alignment, data storage, data TLB error
41	—	Reserved, should be cleared.	—
42	DLK	Cache locking. Settings are implementation-dependent. On the e500, DLK is set when a DSI occurs because dcbtIs , dcbtstIs , or dcblc is executed in user mode and MSR[UCLE] = 0.	Data storage
43	ILK	Set when a DSI occurs because icbtl or icblc is executed in user mode (MSR[PR] = 1) and MSR[UCLE] = 0	Data storage
44	AP	Auxiliary processor operation (not supported on the e500)	Alignment, data storage, data TLB, program
45	PUO	Unimplemented operation exception (not supported on the e500. Unimplemented instructions are handled as illegal instructions.)	Program
46	BO	Byte-ordering exception	Data storage, instruction storage
47	PIE	Imprecise exception (not supported on the e500)	Program
48–55	—	Reserved, should be cleared.	—
56	SPE	SPE exception bit. Because Book E allocates this bit for implementation-dependent use, it may have a different functions on other implementations.	—
57–63	—	Allocated for implementation-dependent use. Reserved, should be cleared.	—

An implementation may define additional ESR bits to identify implementation-specific or architected interrupt types; the e500 implements ESR[ILK] and ESR[SPE].

NOTE

System software may need to identify the type of instruction that caused the interrupt and examine the TLB entry and ESR to fully identify the exception or exceptions. For example, a data storage interrupt may be caused by both a protection violation exception and a byte-ordering exception. System software would have to look beyond ESR[BO], such as the state of MSR[PR] in SRR1 and the TLB entry page protection bits, to determine if a protection violation also occurred.

Table 5-3 shows MCSR bit definitions.

Table 5-3. Machine Check Syndrome Register (MCSR)

Bit	Name	Description	Recoverable
32	MCP	Machine check input pin	Maybe
33	ICPERR	Instruction cache parity error	Precise
34	DCP_PERR	Data cache push parity error	Maybe
35	DCPERR	Data cache parity error	Precise (cache-inhibited stwcx. instructions and guarded loads that initiate this error are deallocated before a machine check is taken. MCSRR0 holds the address of the next instruction not the instruction that initiated the error.)
43–46	—	Reserved, should be cleared.	—
47	GL_CI	Set when a guarded load or cache-inhibited stwcx. provokes a parity error in the cache (ICPERR or DCPERR). In this case, the address saved in MCSRR0 is the address after the instruction that caused the error.	
48	ABIST_IC	Abist instruction cache error	NA
49	ABIST_DC	Abist data cache error	NA
50	ABIST_ITLB	Abist instruction TLB error	NA
51	ABIST_DTLB	Abist data TLB error	NA
52	ABIST_L2TLB	Abist L2 TLB error	NA
53	ABIST_TLB1	Abist TLB1 error	NA
54	ABIST_BTBT	Abist BTB error	NA
56	BUS_IAERR	Bus—instruction address error	Maybe
57	BUS_RAERR	Bus—read address error	Maybe
58	BUS_WAERR	Bus—write address error	Maybe
59	BUS_IBERR	Bus—instruction data bus error	Maybe
60	BUS_RBERR	Bus—read data bus error	Maybe
61	BUS_WBERR	Bus—write bus error	Maybe
62	BUS_IPERR	Bus—instruction parity error	Maybe
63	BUS_RPERR	Bus—read parity error	Maybe

5.4 Exceptions

Exceptions are caused directly by instruction execution or by an asynchronous event. In either case, the exception may cause one of several types of interrupts to be invoked.

The following examples are of exceptions caused directly by instruction execution:

- An attempt to execute a reserved-illegal instruction (illegal instruction exception-type program interrupt)
- An attempt by an application program to execute a privileged instruction or to access a privileged SPR (privileged instruction exception-type program interrupt)
- In general, an attempt by an application program to access a nonexistent SPR (unimplemented operation instruction exception type program interrupt). Note the following behavior on the e500:
 - If MSR[PR] = 1 (user mode), SPRN5 = 0 (user-accessible SPR), and the SPR number is invalid, an illegal instruction exception is taken.
 - If MSR[PR] = 0 (supervisor mode) and the SPR number is invalid, an illegal instruction exception is taken.
 - If MSR[PR] = 1, SPRN5 = 1, and invalid SPR address (supervisor-only SPR), a privileged instruction exception type program interrupt is taken.
- An attempt by a system program to access a nonexistent SPR (boundedly undefined)
- The execution of a defined instruction using an invalid form (illegal instruction exception-type program interrupt, unimplemented operation exception type program interrupt, or privileged instruction exception type program interrupt) The e500 does not support unimplemented operation exceptions. Such conditions are processed as illegal instruction exceptions.
- An attempt to access a location that is either unavailable (instruction or data TLB error interrupt) or not permitted (instruction or data storage interrupt)
- An attempt to access a location with an effective address alignment not supported by the implementation (alignment interrupt)
- Execution of a System Call (**sc**) instruction (system call interrupt)
- Execution of a **trap** instruction whose trap condition is met (trap type program interrupt)
- The execution of a defined instruction that is not implemented (illegal instruction exception or unimplemented operation exception type of program interrupt)
- The execution of an allocated instruction that is not implemented (illegal instruction exception or unimplemented operation exception type of program interrupt)

Invocation of an interrupt is precise, except that if one of the imprecise modes for invoking the floating-point enabled exception type program interrupt (not supported on the e500) is in effect, the invocation may be imprecise. When the interrupt is invoked imprecisely, the executing instruction does not appear to complete before the next instruction starts (because the invocation of the interrupt required to complete execution has not occurred).

5.5 Interrupt Classes

All interrupts except machine check are categorized by two independent characteristics:

- **Critical/noncritical.** Some interrupt types demand immediate attention even if other interrupt types being processed have not had the opportunity to save the machine state (that is, return address and captured state of the MSR). To enable taking a critical interrupt immediately after a noncritical interrupt is taken (that is, before the machine state is saved), two sets of save/restore register pairs are provided. Critical interrupts use CSRR0/CSRR1 and noncritical interrupts use SRR0/SRR1.
- **Asynchronous/synchronous.** Asynchronous interrupts are caused by events external to instruction execution; synchronous interrupts are caused by instruction execution and are either precise or imprecise. Table 5-4 describes asynchronous and synchronous exceptions.

Table 5-4. Asynchronous and Synchronous Interrupts

Class	Description
Asynchronous	Caused by events independent from instruction execution. For asynchronous interrupts, the address reported to the exception handling routine is the address of the instruction that would have executed next, had the asynchronous interrupt not occurred.

Table 5-4. Asynchronous and Synchronous Interrupts (continued)

Class	Description				
Synchronous	Caused directly by instruction execution. Synchronous interrupts are precise or imprecise. <table> <tr> <td>Precise</td><td> These interrupts precisely indicate the address of the instruction causing the exception that generated the interrupt; or, for certain synchronous, precise interrupt types, the address of the immediately following instruction. When the execution or attempted execution of an instruction causes a synchronous, precise interrupt, the following conditions exist at the interrupt point. • SRR0 or CSRR0 addresses either the instruction causing the exception or the instruction immediately following the instruction causing the exception. The instruction addressed is determined from the interrupt type and status bits. • An interrupt is generated such that all instructions preceding the instruction causing the exception appear to have completed with respect to the executing processor. However, some accesses associated with these preceding instructions may not have been performed with respect to other processors and mechanisms. • The exception-causing instruction may appear not to have begun execution (except for causing the exception), may be partially executed, or may have completed, depending on the interrupt type. See Section 5.9, "Partially Executed Instructions." • Architecturally, no subsequent instruction has executed beyond the instruction that caused the exceptions. </td></tr> <tr> <td>Imprecise</td><td> Imprecise interrupts may indicate the address of the instruction causing the exception that generated the interrupt or some instruction after that instruction. When execution or attempted execution of an instruction causes an imprecise interrupt, the following conditions exist at the interrupt point. • SRR0 or CSRR0 addresses either the exception-causing instruction or some instruction following the exception-causing instruction that generated the interrupt. • An interrupt is generated such that all instructions preceding the instruction addressed by SRR0 or CSRR0 appear to have completed with respect to the executing processor. • If context synchronization forces the imprecise interrupt due to an instruction that causes another exception that generates an interrupt (for example, alignment or data storage), SRR0 addresses the interrupt-forcing instruction, which may have partially executed (see Section 5.9, "Partially Executed Instructions"). • If execution synchronization forces an imprecise interrupt due to an execution-synchronizing instruction other than msync or isync, SRR0 or CSRR0 addresses the interrupt-forcing instruction, which appears not to have begun execution (except for its forcing the imprecise interrupt). If the interrupt is forced by msync or isync, SRR0 or CSRR0 may address msync or isync, or the following instruction. • If context or execution synchronization forces an imprecise interrupt, the instruction addressed by SRR0 or CSRR0 may have partially executed (see Section 5.9, "Partially Executed Instructions"). No instruction following the instruction addressed by SRR0 or CSRR0 has executed. </td></tr> </table>	Precise	These interrupts precisely indicate the address of the instruction causing the exception that generated the interrupt; or, for certain synchronous, precise interrupt types, the address of the immediately following instruction. When the execution or attempted execution of an instruction causes a synchronous, precise interrupt, the following conditions exist at the interrupt point. • SRR0 or CSRR0 addresses either the instruction causing the exception or the instruction immediately following the instruction causing the exception. The instruction addressed is determined from the interrupt type and status bits. • An interrupt is generated such that all instructions preceding the instruction causing the exception appear to have completed with respect to the executing processor. However, some accesses associated with these preceding instructions may not have been performed with respect to other processors and mechanisms. • The exception-causing instruction may appear not to have begun execution (except for causing the exception), may be partially executed, or may have completed, depending on the interrupt type. See Section 5.9, "Partially Executed Instructions." • Architecturally, no subsequent instruction has executed beyond the instruction that caused the exceptions.	Imprecise	Imprecise interrupts may indicate the address of the instruction causing the exception that generated the interrupt or some instruction after that instruction. When execution or attempted execution of an instruction causes an imprecise interrupt, the following conditions exist at the interrupt point. • SRR0 or CSRR0 addresses either the exception-causing instruction or some instruction following the exception-causing instruction that generated the interrupt. • An interrupt is generated such that all instructions preceding the instruction addressed by SRR0 or CSRR0 appear to have completed with respect to the executing processor. • If context synchronization forces the imprecise interrupt due to an instruction that causes another exception that generates an interrupt (for example, alignment or data storage), SRR0 addresses the interrupt-forcing instruction, which may have partially executed (see Section 5.9, "Partially Executed Instructions"). • If execution synchronization forces an imprecise interrupt due to an execution-synchronizing instruction other than msync or isync, SRR0 or CSRR0 addresses the interrupt-forcing instruction, which appears not to have begun execution (except for its forcing the imprecise interrupt). If the interrupt is forced by msync or isync, SRR0 or CSRR0 may address msync or isync, or the following instruction. • If context or execution synchronization forces an imprecise interrupt, the instruction addressed by SRR0 or CSRR0 may have partially executed (see Section 5.9, "Partially Executed Instructions"). No instruction following the instruction addressed by SRR0 or CSRR0 has executed.
Precise	These interrupts precisely indicate the address of the instruction causing the exception that generated the interrupt; or, for certain synchronous, precise interrupt types, the address of the immediately following instruction. When the execution or attempted execution of an instruction causes a synchronous, precise interrupt, the following conditions exist at the interrupt point. • SRR0 or CSRR0 addresses either the instruction causing the exception or the instruction immediately following the instruction causing the exception. The instruction addressed is determined from the interrupt type and status bits. • An interrupt is generated such that all instructions preceding the instruction causing the exception appear to have completed with respect to the executing processor. However, some accesses associated with these preceding instructions may not have been performed with respect to other processors and mechanisms. • The exception-causing instruction may appear not to have begun execution (except for causing the exception), may be partially executed, or may have completed, depending on the interrupt type. See Section 5.9, "Partially Executed Instructions." • Architecturally, no subsequent instruction has executed beyond the instruction that caused the exceptions.				
Imprecise	Imprecise interrupts may indicate the address of the instruction causing the exception that generated the interrupt or some instruction after that instruction. When execution or attempted execution of an instruction causes an imprecise interrupt, the following conditions exist at the interrupt point. • SRR0 or CSRR0 addresses either the exception-causing instruction or some instruction following the exception-causing instruction that generated the interrupt. • An interrupt is generated such that all instructions preceding the instruction addressed by SRR0 or CSRR0 appear to have completed with respect to the executing processor. • If context synchronization forces the imprecise interrupt due to an instruction that causes another exception that generates an interrupt (for example, alignment or data storage), SRR0 addresses the interrupt-forcing instruction, which may have partially executed (see Section 5.9, "Partially Executed Instructions"). • If execution synchronization forces an imprecise interrupt due to an execution-synchronizing instruction other than msync or isync, SRR0 or CSRR0 addresses the interrupt-forcing instruction, which appears not to have begun execution (except for its forcing the imprecise interrupt). If the interrupt is forced by msync or isync, SRR0 or CSRR0 may address msync or isync, or the following instruction. • If context or execution synchronization forces an imprecise interrupt, the instruction addressed by SRR0 or CSRR0 may have partially executed (see Section 5.9, "Partially Executed Instructions"). No instruction following the instruction addressed by SRR0 or CSRR0 has executed.				

5.5.1 Requirements for System Reset Generation

Book E does not specify a system reset interrupt as was defined in the PowerPC architecture. A system reset is initiated in one of the following ways:

- By asserting core_hreset, which resets the internal state of the core complex
- By writing a 1 to DBCR0[34], if MSR[DE] = 1

5.6 Interrupt Processing

Associated with each kind of interrupt is an interrupt vector, the address of the initial instruction that is executed when an interrupt occurs.

Interrupt processing consists of saving a small part of the processor's state in certain registers, identifying the cause of the interrupt in another register, and continuing execution at the corresponding interrupt vector location. When an exception exists that causes an interrupt to be generated and it has been determined that the interrupt can be taken, the following steps are performed:

1. SRR0 (for noncritical class interrupts) or CSRR0 (for critical class interrupts) or MCSRR0 for machine check interrupts is loaded with an instruction address that depends on the type of interrupt; see the specific interrupt description for details.
2. The ESR is loaded with information specific to the exception type. Note that many interrupt types can only be caused by a single type of exception event, and thus do not need nor use an ESR setting to indicate the cause of the interrupt.
3. SRR1 (for noncritical class interrupts) or CSRR1 (for critical class interrupts) or MCSRR1 for machine check interrupts is loaded with a copy of the MSR contents.
4. New MSR values take effect beginning with the first instruction following the interrupt. The MSR is updated as follows:
 - MSR[SPE,WE,EE,PR,FP,FE0,FE1,IS,DS] are cleared by all interrupts.
 - MSR[CE,DE] are cleared by critical class interrupts and unchanged by noncritical class interrupts.
 - MSR[ME] is cleared by machine check interrupts and unchanged by other interrupts.
 - Other defined MSR bits are unchanged by all interrupts.

MSR fields are described in Section 2.5.1, “Machine State Register (MSR).”

5. Instruction fetching and execution resumes, using the new MSR value, at a location specific to the interrupt type (IVPR[32–47] || IVOR n [48–59] || 0b0000)

The IVOR n for the interrupt type is described in Table 5-5. IVPR and IVOR contents are indeterminate upon reset and must be initialized by system software.

Interrupts do not clear reservations obtained with load and reserve instructions. The operating system should do so at appropriate points, such as at process switch.

At the end of a noncritical interrupt handling routine, executing **rfi** causes the MSR to be restored from the SRR1 contents and instruction execution to resume at the address contained in SRR0. Likewise, **rfci** performs the same function at the end of a critical interrupt handling routine, using CSRR0 instead of SRR0 and CSRR1 instead of SRR1.

NOTE

In general, at process switch, due to possible process interlocks and possible data availability requirements, the operating system needs to consider executing the following:

- **stwcx**—clears outstanding reservations to prevent pairing a **lwarx** in the old process with a **stwcx** in the new one.
- **msync**—ensures that memory operations of an interrupted process completes with respect to other processors before that process begins executing on another processor.
- **rfi**, **rfci**, **rfmci**, or **isync**—ensures that instructions in the new process execute in the new context.

5.7 Interrupt Definitions

Table 5-5 summarizes each interrupt type, the various exception types that may cause that interrupt, the interrupt classification, which ESR bits can be set, which MSR bits can mask the interrupt type, and which IVOR is used to specify the vector address.

Table 5-5. Interrupt and Exception Types

IVOR	Interrupt Type	Exception Type	Exception Class ¹	ESR ²	Mask Bits	Notes	Page
IVOR0	Critical input	Critical input	A, C		MSR[CE]	³	14
IVOR1	Machine check	Machine check	C		MSR[ME]	^{4, 5}	15
IVOR2	Data storage (DSI)	Access	SP	[SPE],[ST],[FP,AP] ⁶		⁷	17
		Load reserve or store conditional to write-through required location (W = 1)	SP	[ST]		⁷	
		Cache locking	SP	[DLK,ILK],[ST]		⁸	
		Byte ordering	SP	[ST],[FP,AP] ⁶ , BO			
IVOR3	Instruction storage (ISI)	Access	SP				19
		Byte ordering	SP	BO			
IVOR4	External input		A		MSR[EE]	³	20
IVOR5	Alignment		SP	[ST],[FP,AP] ⁶ , [SPE,AP,ST]			21

Table 5-5. Interrupt and Exception Types (continued)

IVOR	Interrupt Type	Exception Type	Exception Class ¹	ESR ²	Mask Bits	Notes	Page
IVOR6	Program	Illegal	SP	PIL			22
		Privileged	SP	PPR,[AP] ⁶			
		Trap	SP	PTR			
		Floating-point enabled (unsupported on the e500)	SP, SI	FP,[PIE]	MSR[FE0,FE1]	⁶	
		Auxiliary processor enabled (unsupported on the e500)	SP	AP ⁶			
		Unimplemented op	SP	PUO,[FP,AP] ⁶		10	
IVOR7	Floating-point unavailable (not supported on the e500)		SP				—
IVOR8	System call		SP				24
IVOR9	Auxiliary processor unavailable (not supported on the e500)		SP				—
IVOR10	Decrementer		A		MSR[EE], TCR[DIE]		24
IVOR11	Fixed interval timer		A		MSR[EE], TCR[FIE]		25
IVOR12	Watchdog		A, C		MSR[CE], TCR[WIE]		26
IVOR13	Data TLB error	Data TLB miss	SP	[SPE],[ST], [FP,AP] ⁶			27
IVOR14	Instruction TLB error	Instruction TLB miss	SP				29
IVOR15	Debug	Trap (synchronous)	A, SP, C		MSR[DE], DBCR0[IDM]		29
		Instruction address compare (synchronous)	A, SP, C		MSR[DE], DBCR0[IDM]		
		Data address compare (synchronous)	A, SP, C		MSR[DE], DBCR0[IDM]		
		Instruction complete	SP, C		MSR[DE], DBCR0[IDM]	⁹	
		Branch taken	SP, C		MSR[DE], DBCR0[IDM]	⁹	
		Return from interrupt	SP, C		MSR[DE], DBCR0[IDM]		
		Interrupt taken	SI, C		MSR[DE], DBCR0[IDM]		
		Unconditional debug event	SI, C		MSR[DE], DBCR0[IDM]		
IVOR32	SPE APU Unavailable	SPE APU unavailable	SP			10	30

Table 5-5. Interrupt and Exception Types (continued)

IVOR	Interrupt Type	Exception Type	Exception Class ¹	ESR ²	Mask Bits	Notes	Page
IVOR33	SPE floating-point data exception	SPE floating-point data exception	SP			10	31
IVOR34	SPE floating-point round exception	SPE floating-point round exception	SP			10	31

¹ A = asynchronous, C = critical, SI = synchronous, imprecise, SP = synchronous, precise

² In general, when an interrupt causes an ESR bit or bits to be set (or cleared) as indicated in the table, it also causes all other ESR bits to be cleared. Special rules may apply for implementation-specific ESR bits.

Legend:

xxx (no brackets) means ESR[xxx] is set.

[xxx] means ESR[xxx] could be set.

{xxx,yyy} means either ESR[xxx] or ESR[yyy] may be set, but never both.

{xxx,yyy} means either ESR[xxx] or ESR[yyy] may be set, or possibly both.

³ Although not part of Book E, system interrupt controllers commonly provide independent mask and status bits for critical input and external input interrupt sources.

⁴ Machine check interrupts are not asynchronous or synchronous. See Section 5.7.2, "Machine Check Interrupt."

⁵ Machine check status information is commonly provided as part of the system implementation but is not part of Book E.

⁶ Not supported in the e500.

⁷ Software must examine the instruction and the subject TLB entry to determine the exact cause of the interrupt.

⁸ Cache locking and cache locking exceptions are implementation-dependent.

⁹ Instruction complete and branch taken debug events are defined only for MSR[DE] = 1 for internal debug mode DBCR0[IDM] = 1. In other words, for internal debug mode with MSR[DE] = 0, instruction complete and branch taken debug events cannot occur, no DCSR status bits are set, and no subsequent imprecise debug interrupt can occur (see Section 8.4, "Book E Debug Events").

¹⁰ e500-specific exception

5.7.1 Critical Input Interrupt

A critical input interrupt occurs when no higher priority exception exists, a critical input exception is presented to the interrupt mechanism, and MSR[CE] = 1. The specific definition of a critical input exception is implementation-dependent but is typically caused by assertion of an asynchronous signal that is part of the system.

CSRR0, CSRR1, and MSR are updated as shown in Table 5-6.

Table 5-6. Critical Input Interrupt Register Settings

Register	Setting
CSRR0	Set to the effective address of the next instruction to be executed.
CSRR1	Set to the MSR contents at the time of the interrupt.
MSR	ME is unchanged. All other MSR bits cleared.

Instruction execution resumes at address $\text{IVPR}[32-47] \parallel \text{IVOR0}[48-59] \parallel 0b0000$.

To guarantee that the core complex can take an critical input interrupt, the critical input interrupt signal must be asserted until the interrupt is taken. Otherwise, whether the core complex takes an external interrupt depends on whether $\text{MSR}[\text{CE}]$ is set while the critical interrupt signal is asserted.

NOTE

To avoid another, redundant critical input interrupt, software is responsible for taking any actions required by the implementation to clear any critical input exception status before reenabling $\text{MSR}[\text{CE}]$.

5.7.2 Machine Check Interrupt

The e500 implements the EIS-defined machine check APU, which differs from the Book E definition as follows:

- Book E defines machine check interrupts as critical class interrupts, but machine check APU treats it as a distinct interrupt type.
- Machine check is no longer a critical interrupt but uses MCSRR0 and MCSRR1 for saving the return address and the MSR in case the machine check is recoverable.
- Return from machine check interrupt instruction (**rfmci**) is implemented to support the return to the address saved in MCSRR0 .
- A machine check syndrome register, MCSR , is used to log the cause of the machine check (instead of ESR). The MCSR is described in Table 5-3.

The following general information applies to both the Book E and EIS definitions. A machine check interrupt occurs when no higher priority exception exists, a machine check exception is presented to the interrupt mechanism, and $\text{MSR}[\text{ME}] = 1$. Specific causes of machine check exceptions are implementation-dependent, as are the details of the actions taken on a machine check interrupt.

Machine check interrupts, which are typically caused by a hardware or memory subsystem failure or by an attempt to access an invalid address, are a special case. A machine check may be caused indirectly by execution of an instruction but may not be recognized or reported until long after the processor has executed past the instruction that caused the

Interrupt Definitions

machine check. As such, machine check interrupts are not thought of as synchronous or asynchronous or as precise or imprecise.

The following general rules apply:

- No instruction after the one whose address is reported to the machine check interrupt handler in MCSRR0 has begun execution.
- The instruction whose address is reported to the machine check interrupt handler in MCSRR0 and all prior instructions may or may not have completed successfully. All instructions certain to complete appear to have done so within the context existing before the machine check interrupt. No further interrupts (other than possible additional machine check interrupts) occur as a result of those instructions.

Machine check exceptions are specified in Table 5-7.

Table 5-7. e500 Machine Check Exception Sources

Source	Additional Enable Bits
Negative edge on machine check signal (MCP)	HID0[32]
Data cache parity error	L1CSR0[DCPE]
Instruction cache parity error	L1CSR1[ICPE]
Data cache push parity error	L1CSR0[DCPE]
Bus instruction address error	No enable bit
Bus read address error	No enable bit
Bus write address error	No enable bit
Bus instruction data bus error	No enable bit
Read data bus error	No enable bit
Write bus error	No enable bit
Instruction parity error	HID1[R1DPE], HID1[R2DPE] (depending on which bus the instruction fetch arrived on)
Read parity error	HID1[R1DPE], HID1[R2DPE] (whichever bus the data read arrived on)

If MSR[ME] is cleared, the e500 enters checkstop state immediately on detecting the machine check condition.

When a machine check interrupt is taken, registers are updated as shown in Table 5-8.

Table 5-8. Machine Check Interrupt Settings

Register	Setting
MCSRR0	32–63 On a best-effort basis, the core complex sets this to an EA of some instruction that was executing or about to be executing when the machine check condition occurred.
MCSRR1	MSR[37–38,46–55,57–59,61–63] are loaded with equivalent MSR bits. All other bits are reserved.
MSR	Cleared: UCLE, SPE, WE, CE, EE, PR, FP, ME, FE0, FE1, RCI, DE, IS, DS, PMM, RI

Table 5-8. Machine Check Interrupt Settings (continued)

Register	Setting
MCAR	When the core complex takes a machine check interrupt, it updates the machine check address register MCAR (see Section 2.7.2.3, “Machine Check Address Register (MCAR)”) to indicate the address of the condition that caused the machine check. Note that if a machine check interrupt is caused by a signal, the contents of MCAR are not meaningful.
MCSR	Set according to the machine check condition. See Table 5-3.
ESR	The e500 uses the MCSR rather than the ESR. Implementation-dependent

Instruction execution resumes at address IVPR[32–47] || IVOR1[48–59] || 0b0000.

NOTES

If a machine check interrupt is caused by a memory subsystem error, the subsystem may return incorrect data, which may be placed into registers or on-chip caches.

For implementations on which a machine check interrupt is caused by referring to an invalid physical address, executing **dcbz** or **dcba** can cause a delayed machine check interrupt by establishing a data cache block associated with an invalid physical address. A machine check interrupt can occur later if and when an attempt is made to write that block to main memory, for example as the result of executing an instruction that causes a cache miss for which the block is the target for replacement or as the result of executing **dcbst** or **dcbf**.

5.7.3 Data Storage Interrupt

A data storage interrupt occurs when no higher priority exception exists and a data storage exception is presented to the interrupt mechanism. Table 5-9 describes exception conditions for a data storage interrupt as defined by Book E.

Table 5-9. Data Storage Interrupt Exception Conditions

Exception	Cause
Read access control exception	Occurs when either of the following conditions exist: - In user mode (MSR[PR] = 1), a load or load-class cache management instruction attempts to access a memory location that is not user-mode read enabled (page access control bit UR = 0). - In supervisor mode (MSR[PR] = 0), a load or load-class cache management instruction attempts to access a location that is not supervisor-mode read enabled (page access control bit SR = 0).
Write access control exception	Occurs when either of the following conditions exist: - In user mode (MSR[PR] = 1), a store or store-class cache management instruction attempts to access a location that is not user-mode write enabled (page access control bit UW = 0). - In supervisor mode (MSR[PR] = 0), a store or store-class cache management instruction attempts to access a location that is not supervisor-mode write enabled (page access control bit SW = 0).

Table 5-9. Data Storage Interrupt Exception Conditions (continued)

Exception	Cause
Byte-ordering exception	The implementation cannot access data in the byte order specified by the page's endian attribute. Note: The byte-ordering exception is provided to assist implementations that cannot support dynamically switching byte ordering between consecutive accesses, the byte order for a class of accesses, or unaligned accesses using a specific byte order. On the e500, load/store accesses that cross a page boundary such that endianness changes cause a byte-ordering exception.
Cache locking exception	The locked state of one or more cache lines has the potential to be altered. This exception is implementation-dependent. On the e500, a cache locking exception occurs with the execution of icbtlts , icblc , dcbtlts , dcbtstlts , or dcblc when (MSR[PR] = 1) and (MSR[UCLC] = 0). ESR is set as follows: - For icbtlts and icblc, ESR[ILK] is set. - For dcbtlts, dcbtstlts, or dcblc, ESR[DLK] is set. Book E refers to this as a cache-locking exception.
Storage synchronization exception	Occurs when either of the following conditions exist: - An attempt is made to execute a load and reserve or store conditional instruction from or to a location that is write-through required or caching inhibited (if the interrupt does not occur, the instruction executes correctly). - A store conditional instruction produces an effective address for which a normal store would cause a data storage interrupt but the processor does not have the reservation from a load and reserve instruction. Book E states that it is implementation-dependent whether a data storage interrupt occurs. The EIS defines that the data storage interrupt is taken. See Section 3.4.1.8, "Atomic Update Primitives Using lwarx and stwcx.."

Instructions **icbt**, **dcbt**, **dcbtst**, and **dcba** cannot cause a data storage interrupt, regardless of the effective address.

NOTE

icbi and **icbt** are treated as loads from the addressed byte with respect to address translation and protection. These instruction cache management instructions use MSR[DS], not MSR[IS], to determine translation for their operands. Instruction storage exceptions and instruction TLB miss exceptions are associated with instruction fetching and not execution. Data storage exceptions and data TLB miss exceptions are associated with the execution of instruction cache management instructions.

When a data storage interrupt occurs, the processor suppresses execution of the instruction causing the data storage exception.

SRR0, SRR1, MSR, DEAR, and ESR are updated as follows:

Table 5-10. Data Storage Interrupt Register Settings

Register	Setting
SRR0	Set to the effective address of the instruction causing the data storage interrupt.
SRR1	Set to the MSR contents at the time of the interrupt.
MSR	CE, ME, and DE are unchanged. All other MSR bits cleared.

Table 5-10. Data Storage Interrupt Register Settings (continued)

Register	Setting
DEAR	Set to the effective address of a byte that lies both within the range of bytes being accessed by the access or cache management instruction and within the page whose access caused the exception.
ESR	FP (Not supported on the e500) Set if the instruction causing the interrupt is a floating-point load or store; otherwise cleared. ST Set if the instruction causing the interrupt is a store or store-class cache management instruction; otherwise cleared. DLK On the e500, DLK is set when a DSI occurs because dcbtIs, dcbtstIs, or dcbIc is executed in user mode and MSR[UCLE] = 0. AP (Not supported on the e500) Set if the instruction causing the interrupt is an auxiliary processor load or store; otherwise cleared. BO Set if the instruction caused a byte-ordering exception; otherwise cleared. All other defined ESR bits are cleared.

Instruction execution resumes at address IVPR[32–47] || IVOR2[48–59] || 0b0000.

5.7.4 Instruction Storage Interrupt

An instruction storage interrupt occurs when no higher priority exception exists and an instruction storage exception is presented to the interrupt mechanism. Instruction storage exception conditions are described in Table 5-11.

Table 5-11. Instruction Storage Interrupt Exception Conditions

Exception	Cause
Execute access control exception	In user mode, an instruction fetch attempts to access a memory location that is not user mode execute enabled (page access control bit UX = 0). In supervisor mode, an instruction fetch attempts to access a memory location that is not supervisor mode execute enabled (page access control bit SX = 0).
Byte-ordering exception	The implementation cannot fetch the instruction in the byte order specified by the page's endian attribute. On the e500, accesses that cross a page boundary such that endianness changes cause a byte-ordering exception.

Note that Book E provides this exception to assist implementations that cannot dynamically switch byte ordering between consecutive accesses, do not support the byte order for a class of accesses, or do not support unaligned accesses using a specific byte order.

When an instruction storage interrupt occurs, the processor suppresses execution of the instruction causing the exception.

SRR0, SRR1, MSR, and ESR are updated as shown in Table 5-12.

Table 5-12. Instruction Storage Interrupt Register Settings

Register	Setting
SRR0	Set to the effective address of the instruction causing the instruction storage interrupt.
SRR1	Set to the MSR contents at the time of the interrupt.

Table 5-12. Instruction Storage Interrupt Register Settings (continued)

Register	Setting
MSR	CE, ME, and DE are unchanged. All other MSR bits cleared.
ESR	BO is set if the instruction fetch caused a byte-ordering exception; otherwise cleared. All other defined ESR bits are cleared.

NOTE

Permissions violation and byte-ordering exceptions are not mutually exclusive. Even if ESR[BO] is set, system software must examine the TLB entry accessed by the instruction fetch to determine whether a permissions violation also may have occurred.

Instruction execution resumes at address IVPR[32–47] || IVOR3[48–59] || 0b0000.

5.7.5 External Input Interrupt

An external input interrupt occurs when no higher priority exception exists, an external input exception is presented to the interrupt mechanism, and MSR[EE] = 1. The specific definition of an external input exception is implementation-dependent and is typically caused by the activation of an asynchronous signal that is part of the processing system. On the e500, this is the external interrupt signal.

To guarantee that the core complex can take an external interrupt, the external interrupt pin must be asserted until the interrupt is taken. Otherwise, whether the core complex takes an external interrupt depends on whether MSR[EE] is set while the external interrupt signal is asserted.

In addition to MSR[EE], implementations may provide other ways to mask the external input interrupt. The e500 does not support additional masking mechanisms.

SRR0, SRR1, and MSR are updated as shown in Table 5-13.

Table 5-13. External Input Interrupt Register Settings

Register	Setting
SRR0	Set to the effective address of the next instruction to be executed.
SRR1	Set to the MSR contents at the time of the interrupt.
MSR	CE, ME, and DE are unchanged. All other MSR bits cleared.

Instruction execution resumes at address IVPR[32–47] || IVOR4[48–59] || 0b0000.

NOTE

Software is responsible for taking any actions required to clear any external input exception status before reenabling MSR[EE] to avoid another, redundant external input interrupt.

5.7.6 Alignment Interrupt

An alignment interrupt occurs when no higher priority exception exists and an alignment exception is presented to the interrupt mechanism. An alignment exception may occur when an implementation cannot perform a data access for one of the following reasons:

- The operand of a load or store is not aligned.
- The instruction is a move assist, load multiple, or store multiple.
- The operand of **dcbz** is in write-through required or caching inhibited memory, or **dcbz** is executed in an implementation that has no data cache or a write-through data cache.
- The operand of a store, except store conditional, is in write-through required memory.

On the e500, any of the following conditions cause an alignment exception:

- Execution of a **dcbz** references a page marked as write-through or cache inhibited.
- A load multiple word instruction (**lmw**) reads an address that is not a multiple of 4.
- A **lwarx** or **stwcx** instruction references an address that is not a multiple of 4.
- SPE APU instructions are not aligned on a natural boundary. A natural boundary is defined by the size of the data element being accessed.
- A vector operation takes an exception if the physical address of the following instructions is not aligned to the 64-bit boundary: **evladd**, **evlddx**, **evldw**, **evldwx**, **evldh**, **evldhx**, **evstdd**, **evstdx**, **evstdw**, **evstdwx**, **evstdh**, and **evstdhx**. Table 5-14 describes additional ESR settings.

For **lmw** and **stmw** with a non-word-aligned operand and for load and reserve and store conditional instructions with an unaligned operand, an implementation may yield boundedly undefined results instead of causing an alignment interrupt. A store conditional to a write-through required location may either cause an alignment or data storage interrupt or may correctly execute the instruction. For all other cases listed above, an implementation may execute the instruction correctly instead of causing an alignment interrupt. For **dcbz**, correct execution means clearing each byte of the block in main memory.

NOTE

Book E does not support the use of an unaligned effective address by load and reserve and store conditional instructions. If one of these instructions specifies an unaligned effective address, the alignment interrupt handler should treat the instruction as a programming error and must not attempt to emulate the instruction.

When an alignment interrupt occurs, the processor suppresses the execution of the instruction causing the alignment exception.

SRR0, SRR1, MSR, DEAR, and ESR are updated as shown in Table 5-14.

Table 5-14. Alignment Interrupt Register Settings

Register	Setting
SRR0	Set to the effective address of the instruction causing the alignment interrupt.
SRR1	Set to the MSR contents at the time of the interrupt.
MSR	CE, ME, and DE are unchanged. All other MSR bits cleared.
DEAR	Set to the EA of a byte that is both within the range of the bytes being accessed by the memory access or cache management instruction, and within the page whose access caused the alignment exception.
ESR	The following bits may be affected on the e500 for vector alignment exception conditions: SPE Set ST Set only if the instruction causing the exception is a store and is cleared for a load. All other defined ESR bits are cleared.

Instruction execution resumes at address $\text{IVPR}[32-47] \parallel \text{IVOR5}[48-59] \parallel 0b0000$.

5.7.7 Program Interrupt

A program interrupt occurs when no higher priority exception exists and a program exception is presented to the interrupt mechanism. A program exception is caused when any of the following exceptions arises during execution of an instruction.

Table 5-15. Program Interrupt Exception Conditions

Exception	Cause
Floating-point enabled exception	(Not supported on the e500.) This exception is caused when $(\text{MSR}[\text{FE0}] \mid \text{MSR}[\text{FE1}]) \& \text{FPSCR}[\text{FEX}] = 1$. $\text{FPSCR}[\text{FEX}]$ is set by the execution of a floating-point instruction that causes an enabled exception, including the case of a Move to FPSCR instruction that causes an exception bit and the corresponding enable bit both to be 1. Note that in this context, the term 'enabled exception' refers to the enabling provided by FPSCR control bits. Whether the interrupt is precise or imprecise is determined by $\text{MSR}[\text{FE0}, \text{FE1}]$, as described in Table 2-7.
Auxiliary processor enabled exception	(Not supported on the e500.) Implementation-dependent.

Table 5-15. Program Interrupt Exception Conditions (continued)

Exception	Cause
Illegal instruction exception	An illegal instruction exception always occurs when execution of any of the following kinds of instructions is attempted. • A reserved-illegal instruction • In user mode, an mtspr or mfspr that specifies an SPRN value with SPRN[5] = 0 (user-mode accessible) that represents an unimplemented SPR • If an invalid SPR address is accessible only in supervisor mode and the e500 is in supervisor mode (MSR[PR] = 0), results are undefined. • If the invalid SPR address is accessible only in the supervisor mode and the processor is in user mode (MSR[PR] = 1), a privileged instruction exception is taken. An illegal instruction exception may occur when execution is attempted of any of the following kinds of instructions. If the exception does not occur, the alternative is shown in parentheses. • An instruction that is in invalid form (boundedly undefined results) • A reserved no-op instruction (no-operation performed is preferred) There are currently no reserved no-ops for the e500. • A defined or allocated instruction that is not implemented (unimplemented operation exception). On the e500, unimplemented Book E instructions such as mfapidi, mfdcr, and mtdcr take an illegal instruction exception. • The e500 takes an illegal instruction exception on an attempt to execute a 64-bit Book E instruction.
Privileged instruction exception	Occurs when MSR[PR] = 1 and execution is attempted of any of the following: • A privileged instruction • An mtspr or mfspr instruction that specifies a privileged SPR (SPRN[5] = 1). • An mtpmr or mfpmr instruction that specifies a privileged PMR (PMRN[5] = 1).
Trap exception	A trap exception occurs when any of the conditions specified in a Trap instruction are met.
Unimplemented operation exception	An unimplemented operation exception may occur when a defined or allocated instruction is encountered that is not implemented. Otherwise an illegal instruction exception occurs. On the e500, these instructions are mfapidi , mfdcr , and mtdcr and they take an illegal instruction exception.

SRR0, SRR1, MSR, and ESR are updated as shown in Table 5-16.

Table 5-16. Program Interrupt Register Settings

Register	Description
SRR0	For all program interrupts except an enabled exception when in an imprecise mode (see Table 5-17), set to the EA of the instruction that caused the interrupt.
SRR1	Set to the MSR contents at the time of the interrupt.

Table 5-16. Program Interrupt Register Settings (continued)

Register	Description
MSR	CE, ME, and DE are unchanged. All other MSR bits cleared.
ESR	PIL Set if an illegal instruction exception type program interrupt; otherwise cleared. PPR Set if a privileged instruction exception type program interrupt; otherwise cleared. PTR Set if a trap exception type program interrupt; otherwise cleared. PUO (Not supported on the e500) Set if an unimplemented operation exception type program interrupt; otherwise cleared. FP (Not supported on the e500) Set if the instruction causing the interrupt is a floating-point instruction; otherwise cleared. PIE (Not supported on the e500) Set if a floating-point enabled exception type program interrupt, and the address saved in SRR0 is not the address of the instruction causing the exception (that is, the instruction that caused FPSCR[FEX] to be set); otherwise cleared. AP (Not supported on the e500) Set if the instruction causing the interrupt is an auxiliary processor instruction; otherwise cleared. All other defined ESR bits are cleared.

Instruction execution resumes at address IVPR[32–47] || IVOR6[48–59] || 0b0000.

5.7.8 System Call Interrupt

A system call interrupt occurs when no higher priority exception exists and a System Call (sc) instruction is executed.

SRR0, SRR1, and MSR are updated as shown in Table 5-17.

Table 5-17. System Call Interrupt Register Settings

Register	Description
SRR0	Set to the effective address of the instruction after the sc instruction.
SRR1	Set to the MSR contents at the time of the interrupt.
MSR	CE, ME, and DE are unchanged. All other MSR bits cleared.

Instruction execution resumes at address IVPR[32–47] || IVOR8[48–59] || 0b0000.

5.7.9 Decrementer Interrupt

A decrementer interrupt occurs when no higher priority exception exists, a decrementer exception exists (TSR[DIS] = 1), and the interrupt is enabled (TCR[DIE] = 1 and MSR[EE] = 1). See Section 9.3.1, “Decrementer Register.”

NOTE

MSR[EE] also enables external input and fixed-interval timer interrupts.

SRR0, SRR1, MSR, and TSR are updated as shown in Table 5-18.

Table 5-18. Decrementer Interrupt Register Settings

Register	Setting
SRR0	Set to the effective address of the next instruction to be executed.
SRR1	Set to the MSR contents at the time of the interrupt.
MSR	CE, ME, and DE are unchanged. All other MSR bits cleared.
TSR	DIS is set. See Section 9.2.3, "Timer Status Register (TSR)."

Instruction execution resumes at address IVPR[32–47] || IVOR10[48–59] || 0b0000.

NOTE

Software is responsible for clearing the decrementer exception status before reenabling MSR[EE] to avoid another redundant decrementer interrupt. To clear the decrementer exception, the interrupt handling routine must clear TSR[DIS]. Clearing is done by writing a word to TSR using **mtspr** with a 1 in any bit position that is to be cleared and 0 in all other bit positions. The write-data to the TSR is not direct data, but is a mask. Writing a 1 to this bit causes it to be cleared; writing a 0 has no effect.

5.7.10 Fixed-Interval Timer Interrupt

A fixed-interval timer interrupt occurs when no higher priority exception exists, a fixed-interval timer exception exists (TSR[FIS] = 1), and the interrupt is enabled (TCR[FIE] = 1 and MSR[EE] = 1). See Section 9.4, "Fixed-Interval Timer."

The fixed-interval timer period is determined by TCR[FP], which when concatenated with TCR[FPEXT], specifies one of 64 bit locations of the time base used to signal a fixed-interval timer exception on a transition from 0 to 1.

TCR[FPEXT],TCR[FP] = 000000 selects TBU[32]

TCR[FPEXT],TCR[FP] = 111111 selects TBL[63]

NOTE

MSR[EE] also enables external input and decrementer interrupts.

SRR0, SRR1, MSR, and TSR are updated as shown in Table 5-19.

Table 5-19. Fixed-Interval Timer Interrupt Register Settings

Register	Setting
SRR0	Set to the effective address of the next instruction to be executed.
SRR1	Set to the MSR contents at the time of the interrupt.

Table 5-19. Fixed-Interval Timer Interrupt Register Settings (continued)

Register	Setting
MSR	CE, ME, and DE are unchanged; all other MSR bits cleared.
TSR	FIS is set. See Section 9.2.3, “Timer Status Register (TSR).”

Instruction execution resumes at address IVPR[32–47] || IVOR11[48–59] || 0b0000.

NOTE

Software is responsible for clearing the fixed-interval timer exception status before reenabling MSR[EE] to avoid another redundant fixed-interval timer interrupt. To clear the fixed-interval timer exception, the interrupt handling routine must clear TSR[FIS]. Clearing is done by writing a word to TSR using **mtspr** with a 1 in any bit position that is to be cleared and 0 in all other bit positions. The write-data to the TSR is not direct data, but a mask. Writing a 1 causes the bit to be cleared, writing a 0 has no effect.

5.7.11 Watchdog Timer Interrupt

A watchdog timer interrupt occurs when no higher priority exception exists, a watchdog timer exception exists (TSR[WIS] = 1), and the interrupt is enabled (TCR[WIE] = 1 and MSR[CE] = 1). See Section 9.5, “Watchdog Timer.”

NOTE

MSR[CE] also enables the critical input interrupt.

CSRR0, CSRR1, MSR, and TSR are updated as shown in Table 5-20.

Table 5-20. Watchdog Timer Interrupt Register Settings

Register	Setting
CSRR0	Set to the effective address of the next instruction to be executed.
CSRR1	Set to the MSR contents at the time of the interrupt.
MSR	ME is unchanged; all other MSR bits cleared.
TSR	WIS is set. See Section 9.2.3, “Timer Status Register (TSR).”

Instruction execution resumes at address IVPR[32–47] || IVOR12[48–59] || 0b0000.

NOTE

Software is responsible for clearing the watchdog timer exception status before reenabling MSR[CE] to avoid another redundant watchdog timer interrupt. To clear the watchdog timer exception, the interrupt handling routine must clear TSR[WIS]. Clearing is done by writing a word to TSR using **mtspr** with a 1 in any bit position that is to be cleared and 0 in all other bit positions. The write-data to the TSR is not direct data, but a mask. Writing a 1 to this bit causes it to be cleared; writing a 0 has no effect.

5.7.12 Data TLB Error Interrupt

A data TLB error interrupt occurs when no higher priority exception exists and the instruction TLB error exception described in Table 5-23, is presented to the interrupt mechanism.

Table 5-21. Instruction TLB Error Interrupt Exception Conditions

Exception	Description
TLB miss exception	Caused when the virtual addresses associated with an instruction fetch do not match any valid entry in the TLB.

If a store conditional instruction produces an effective address for which a normal store would cause a data TLB error interrupt, but the processor does not have the reservation from a load and reserve instruction, it is implementation-dependent whether a data TLB error interrupt occurs. The EIS defines that the data TLB interrupt is taken in this case.

When a data TLB error interrupt occurs, the processor suppresses execution of the instruction causing the data TLB error exception.

SRR0, SRR1, MSR, DEAR, and ESR are updated as shown in Table 5-22.

Table 5-22. Data TLB Error Interrupt Register Settings

Register	Setting
SRR0	Set to the effective address of the instruction causing the data TLB error interrupt.
SRR1	Set to the MSR contents at the time of the interrupt.
MSR	CE, ME, and DE are unchanged. All other MSR bits cleared.
DEAR	Set to the EA of a byte that is both within the range of the bytes being accessed by the memory access or cache management instruction and within the page whose access caused the data TLB error exception.

Table 5-22. Data TLB Error Interrupt Register Settings (continued)

Register	Setting
ESR	ST Set if the instruction causing the interrupt is a store, dcbi , or dcbz instruction; otherwise cleared. FP (Not supported on the e500) Set if the instruction causing the interrupt is a floating-point load or store; otherwise cleared. AP (Not supported on the e500) Set if the instruction causing the interrupt is an auxiliary processor load or store; otherwise cleared. All other defined ESR bits are cleared.
MASn	See Table 5-23.

Table 5-23 shows MAS register settings for data and instruction TLB error interrupts as implemented on the e500. Section 11.3.6.3, “MAS Register Update Summary for Exceptions, tlbsx, and tlbre,” describes how these values are determined as defined by the EIS.

Table 5-23. MMU Assist Register Field Updates for TLB Error Interrupts

MAS Register Bit/Field	Value Loaded for Each Case
TLBSEL	TLBSELD
ESEL	if TLBSELD = 0: TLB0[NV] else, undefined
NV	if TLBSELD = 0: ~TLB0[NV] else, undefined
V	1
IProt	0
TID[0–7]	Value of PID register selected by TIDSELD
TS	MSR[IS/DS]
TSIZE[0–3]	TSIZED
EPN[32–51]	EPN of access
X0, X1 WIMGE	X0D, X1D WIMGED
RPN[32–51]	Zeroes
PERMIS	Zeroes
TLBSELD	—
TIDSELD[0–1]	—
TSIZED[0–3]	—
WIMGED	—
SPID0	PID0
SAS	MSR[IS] for instruction access; MSR[DS] for data access
SHAREN	SHAREND

Instruction execution resumes at address IVPR[32–47] || IVOR13[48–59] || 0b0000.

5.7.13 Instruction TLB Error Interrupt

An instruction TLB error interrupt occurs when no higher priority exception exists and the instruction TLB error exception described in Table 5-24, is presented to the interrupt mechanism.

Table 5-24. Instruction TLB Error Interrupt Exception Conditions

Exception	Description
TLB miss exception	Caused when the virtual addresses associated with an instruction fetch do not match any valid entry in the TLB.

When an instruction TLB error interrupt occurs, the processor suppresses the execution of the instruction causing the instruction TLB miss exception.

SRR0, SRR1, and MSR are updated as shown in Table 5-25.

Table 5-25. Instruction TLB Error Interrupt Register Settings

Register	Setting
SRR0	Set to the effective address of the instruction causing the instruction TLB error interrupt.
SRR1	Set to the MSR contents at the time of the interrupt.
MSR	CE, ME, and DE are unchanged. All other MSR bits cleared.
MASn	See Table 5-23.

Instruction execution resumes at address IVPR[32–47] || IVOR14[48–59] || 0b0000.

5.7.14 Debug Interrupt

A debug interrupt occurs when no higher priority exception exists, a debug exception exists in the DBSR, and debug interrupts are enabled (DBCR0[IDM] = 1 and MSR[DE] = 1). A debug exception occurs when a debug event causes a corresponding DBSR bit to be set. See Chapter 8, “Debug Support.”

Note that on the e500, if DBCR0[IDM] is cleared, no debug events occur. That is, irrespective of MSR, DBCR0, DBCR1, and DBCR2 settings, no debug events are logged in DBSR and no debug interrupts are taken. If DBCR0[IDM] is set, Book E debug mode functions as specified in Book E (according to the value of MSR[DE] and the values of DBCR0, DBCR1, and DBCR2).

The e500 core complex complies with the Book E debug definition, except as follows:

- Data address compare is only supported for effective addresses.
- Instruction address compares IAC3 and IAC4 are not supported.

Interrupt Definitions

- Instruction address compare is only supported for effective addresses.
- DVC is not supported.

CSRR0, CSRR1, MSR, and DBSR are updated as shown in Table 5-26.

Table 5-26. Debug Interrupt Register Settings

Register	Setting
CSRR0	For debug exceptions that occur while debug interrupts are enabled (DBCR0[IDM] = 1 and MSR[DE] = 1), CSRR0 is set as follows: • For instruction address compare (IAC1 and IAC2), data address compare (DAC1R, DAC1W, DAC2R, and DAC2W), trap (TRAP), or branch taken (BRT) debug exceptions, set to the address of the instruction causing the debug interrupt. • For instruction complete (ICMP) debug exceptions, set to the address of the instruction that would have executed after the one that caused the debug interrupt. • For unconditional debug event (UDE) debug exceptions, set to the address of the instruction that would have executed next if the debug interrupt had not occurred. • For interrupt taken (IRPT) debug exceptions, set to the interrupt vector value of the interrupt that caused the interrupt taken debug event. • For return from interrupt (RET) debug exceptions, set to the address of the instruction that would have executed after the rfi or rfci that caused the debug interrupt. • For debug exceptions that occur while debug interrupts are disabled (DBCR0[IDM] = 0 or MSR[DE] = 0), a debug interrupt occurs at the next synchronizing event if DBCR0[IDM] and MSR[DE] are modified such that they are both set and if the debug exception status is still set in the DBSR. When this occurs, CSRR0 is set to the address of the instruction that would have executed next, not with the address of the instruction that modified the debug control register 0 or MSR and thus caused the interrupt.
CSRR1	Set to the MSR contents at the time of the interrupt.
MSR	ME is unchanged. All other MSR bits are cleared.
DBSR	Set to indicate type of debug event (see Chapter 8, "Debug Support").

Instruction execution resumes at address IVPR[32–47] || IVOR15[48–59] || 0b0000.

5.7.15 e500-Specific Exceptions

The following three exceptions are specific to the e500.

5.7.15.1 SPE APU Unavailable Exception

An SPE APU unavailable exception is taken if MSR[SPE] is cleared and an SPE instruction is executed. When an SPE unavailable exception occurs, the processor suppresses execution of the instruction causing the exception. Table 5-27 describes register settings.

Table 5-27. SPE APU Unavailable Exception Register Settings

Register	Setting
SRR0	Set to the effective address of the instruction causing the exception.
SRR1	Set to the MSR contents at the time of the exception.

Table 5-27. SPE APU Unavailable Exception Register Settings

Register	Setting
MSR	CE, ME, and DE are unchanged. All other bits are cleared.
ESR	SPE (bit 24) is set. All other ESR bits are cleared.

Instruction execution resumes at address IVPR[032–47] || IVOR32[48–59] || 0b0000.

5.7.15.2 SPE Floating-Point Data Exception

A floating-point data exception is generated in the following cases:

- SPEFSCR[FINVE] = 1 and either SPEFSCR[FINVH,FINV] = 1
- SPEFSCR[FDBZE] = 1 and either SPEFSCR[FDBZH,FDBZ] = 1
- SPEFSCR[FUNFE] = 1 and either SPEFSCR[FUNFH,FUNF] = 1
- SPEFSCR[FOVFE] = 1 and either SPEFSCR[FOVFH,FOVF] = 1

Note that although SPEFSCR status bits can be updated with an **mtspr[SPEFSCR]**, exceptions occur only if they are set as the result of an arithmetic operation.

When an SPE floating-point data exception occurs, the processor suppresses execution of the instruction causing the exception. Table 5-28 shows register settings.

Table 5-28. SPE Floating-Point Data Exception Register Settings

Register	Setting
SRR0	Set to the effective address of the instruction causing the exception.
SRR1	Set to the MSR contents at the time of the exception.
MSR	CE, ME, and DE are unchanged. All other bits are cleared.
ESR	SPE (bit 24) is set. All other ESR bits are cleared.
SPEFSCR	One or more of the FINVH, FINV, FDBZH, FDBZ, FUNFH, FUNF, FOVFH, or FOVF bits are set to indicate the exception type.

Instruction execution resumes at address IVPR[032–47] || IVOR33[48–59] || 0b0000.

5.7.15.3 SPE Floating-Point Round Exception

The SPE floating-point round exception is taken if any of the following conditions is met:

- SPEFSCR[FINXE] = 1 and any of the SPEFSCR[FGH,FXH,FG,FX] bits = 1
- SPEFSCR[FRMC] = 0b10 (+∞)
- SPEFSCR[FRMC] = 0b11 (−∞)

Note that although these SPEFSCR status bits can be updated with an **mtspr[SPEFSCR]**, exceptions occur only if they are set as the result of an arithmetic operation.

When an SPE floating-point round exception occurs, the unrounded (truncated) result is placed in the target registers. Table 5-29 describes register settings.

Table 5-29. SPE Floating-Point Round Exception Register Settings

Register	Setting
SRR0	Set to the effective address of the instruction following the instruction causing the exception.
SRR1	Set to the MSR contents at the time of the exception.
MSR	CE, ME, and DE are unchanged. All other MSR bits are cleared.
ESR	SPE (bit 24) is set. All other ESR bits are cleared.
SPEFSCR	FGH, FXH, FG, FX, and FRMC are set appropriately to indicate the exception type.

Instruction execution resumes at address `IVPR[032–47] || IVOR34[48–59] || 0b0000`.

5.8 Performance Monitor Interrupt

The performance monitor provides a performance monitor interrupt that is triggered by an enabled condition or event. An enabled condition or event is as follows:

A PMCx register overflow condition occurs with the following settings:

- `PMLCax[CE] = 1`, that is for the given counter the overflow condition is enabled
- `PMCx[32] = 1`, that is the given counter indicates an overflow

For a performance monitor exception to be signaled on an enabled condition or event, `PMGC0[PMIE]` must be set.

The performance monitor can also freeze the performance monitor counters triggered by an enabled condition or event. For the performance monitor counters to freeze on an enabled condition or event, `PMGC0[FCECE]` must be set.

Although the exception condition could occur with `MSR[EE] = 0`, the interrupt cannot be taken until `MSR[EE] = 1`. If a counter overflows while `PMGC0[FCECE] = 0`, `PMLCa[CE] = 1`, and `MSR[EE] = 0`, it is possible for the counter to wrap around to all zeros again without the performance monitor interrupt being taken.

The priority of the performance monitor interrupt is below the fixed-interval interrupt and above the decremter interrupt. For details on the performance monitor interrupt, see Chapter 7, “Performance Monitor.”

5.9 Partially Executed Instructions

In general, the architecture permits load and store instructions to be partially executed, interrupted, and then restarted from the beginning upon return from the interrupt. To guarantee that a particular load or store instruction completes without being interrupted and

restarted, software must mark the memory as guarded and must use an elementary (non-string or non-multiple) load or store aligned on an operand-sized boundary.

To guarantee that load and store instructions can, in general, be restarted and completed correctly without software intervention, the following rules apply when an execution is partially executed and then interrupted:

- For an elementary load, no part of a target register **rD** has been altered.
- For update forms of load or store, the update register, **rA**, will not have been altered.

The following effects are permissible when certain instructions are partially executed and then restarted:

- For any store, some of the bytes at the target location may have been altered (if write access to that page in which bytes were altered is permitted by the access control mechanism). In addition, for store conditional instructions, **CR0** has been set to an undefined value, and it is undefined whether the reservation has been cleared.
- For any load, some bytes at the addressed location may have been accessed (if read access to that page in which bytes were accessed is permitted by the access control mechanism).
- For load multiple or load string, some registers in the range to be loaded may have been altered. Including the addressing registers **rA** and possibly **rB** in the range to be loaded is a programming error, and thus the rules for partial execution do not protect against overwriting of these registers.

In no case is access control violated.

As previously stated, elementary, aligned, guarded loads and stores are the only access instructions guaranteed to not be interrupted after being partially executed. The following list identifies the specific instruction types for which interruption after partial execution may occur, as well as the specific interrupt types that could cause the interruption:

1. Any load or store (except elementary, aligned, guarded):
 - Any asynchronous interrupt
 - Machine check
 - Decrementer
 - Fixed-interval timer
 - Watchdog timer
 - Debug (unconditional debug event)
2. Unaligned elementary load or store, or any multiple or string:

All of the above listed under item 1, plus the following:

 - Alignment
 - Data storage (if the access crosses a protection boundary)
 - Debug (data address compare)

5.10 Interrupt Ordering and Masking

Multiple exceptions that can each generate an interrupt can exist simultaneously. However, the architecture does not provide for reporting multiple simultaneous interrupts of the same class (critical or noncritical). Therefore, the architecture defines that interrupts are ordered with one another and provides a way to mask certain persistent interrupt types.

When an interrupt type is masked (disabled) and an event causes an exception that would normally generate an interrupt of that type, the exception persists as a status bit in a register (which register depends upon the exception type) but no interrupt is generated. Later, if the interrupt type is enabled (unmasked) and the exception status has not been cleared by software, the interrupt due to the original exception event is finally generated. (The e500 only has such a mechanism for certain debug events. A signal that triggers an asynchronous interrupt, such as external input, must be asserted until they are taken. There is no mechanism for saving the external interrupt if the signal is negated before the interrupt is taken. All interrupts are level-sensitive except for machine check, which is edge-triggered.)

All asynchronous interrupt types and some synchronous interrupt types can be masked. An example of a maskable synchronous interrupt type is the floating-point enabled exception type program interrupt (not supported on the e500). The execution of a floating-point instruction that causes FPSCR[FEX] to be set is considered an exception event, regardless of the setting of MSR[FE0,FE1]. If MSR[FE0,FE1] are both 0, the floating-point enabled exception type program interrupt is masked, but the exception persists in FPSCR[FEX]. Later, if MSR[FE0,FE1] are enabled, the interrupt is generated.

The architecture allows implementations to avoid situations in which an interrupt would cause state information (saved in save/restore registers) from a previous interrupt to be overwritten and lost. As a first step, upon any noncritical class interrupt, hardware automatically disables further asynchronous, noncritical class interrupts (external input) by clearing MSR[EE]. Likewise, upon any critical class interrupt, hardware automatically disables further asynchronous interrupts, both critical and noncritical, by clearing MSR[CE] and MSR[EE]. Critical input, watchdog timer, and debug interrupts are disabled by clearing MSR[CE,DE]. Note that machine check interrupts, while considered neither asynchronous nor synchronous, are not maskable by MSR[CE,DE,EE] and could be presented in a situation that could cause loss of state information.

This first step of clearing MSR[EE] (and MSR[CE,DE] for critical class interrupts) prevents subsequent asynchronous interrupt from overwriting save/restore registers before software can save their contents. On any interrupt, hardware also automatically clears MSR[WE,PR,FP,FE0,FE1,IS,DS], which helps avoid subsequent interrupts of certain other types. However, guaranteeing that these interrupt types do not occur also requires system software to avoid executing instructions that could cause (or enable) a subsequent interrupt, if SRR1 contents have not been saved.

5.10.1 Guidelines for System Software

Table 5-30 lists actions system software must avoid before saving save/restore register contents.

Table 5-30. Operations to Avoid

Operation	Reason
Reenabling MSR[EE] (or MSR[CE,DE] in critical class interrupt handlers)	Prevents any asynchronous interrupts, as well as (in the case of MSR[DE]) any debug interrupts, including synchronous and asynchronous types.
Branching (or sequential execution) to addresses not mapped by the TLB, mapped without UX = 1 or SX = 1 permission, or causing large address or instruction address overflow exceptions.	Prevents instruction storage, instruction TLB error, and instruction address overflow interrupts.
Load, store, or cache management instructions to addresses not mapped by the TLB, causing large address exceptions, or not having required access permissions.	Prevents data storage and data TLB error interrupts.
Execution of System Call (sc) or trap (tw, twi, td, tdi) instructions	Prevents system call and trap exception type program interrupts.
Reenabling of MSR[PR]	Prevents privileged instruction exception type program interrupts. Alternatively, software could reenale MSR[PR] but avoid executing any privileged instructions.
Execution of any illegal instructions	Prevents illegal instruction exception type program interrupts.
Execution of any instruction that could cause an alignment interrupt	Prevents alignment interrupts, including string or multiple instructions and unaligned elementary load or store instructions. Section 5.7.6, "Alignment Interrupt," lists instructions that cause alignment interrupts.

It is not necessary for hardware or software to avoid critical class interrupts from within noncritical class interrupt handlers (and hence hardware does not automatically clear MSR[CE,ME,DE] upon a noncritical interrupt), since the two classes of interrupts use different pairs of save/restore registers to save the instruction address and MSR. The converse, however, is not true. That is, hardware and software must cooperate to avoid both critical and noncritical class interrupts from within critical class interrupt handlers, even though the two classes of interrupts use different SRR pairs. This is because the critical class interrupt may have occurred from within a noncritical handler, before the noncritical handler having saved the SRR0/SRR1. Therefore, within the critical class interrupt handler, both pairs of save/restore registers may contain data necessary to system software.

5.10.2 Interrupt Order

Enabled interrupt types for which simultaneous exceptions can exist are prioritized as follows:

1. Synchronous (non-debug) interrupts:
 - Data storage

Exception Priorities

- Instruction storage
- Alignment
- Program
- System call
- Data TLB error
- Instruction TLB error

Only one of the above types of synchronous interrupts may have an existing exception generating it at a given time. This is guaranteed by the exception priority mechanism (see Section 5.11, “Exception Priorities”) and the requirements of the sequential execution model.

2. Machine check
3. Debug
4. Critical input
5. Watchdog timer
6. External input
7. Fixed-interval timer
8. Decrementer

Although, as indicated above, noncritical, synchronous exception types listed under item 1 are generated with higher priority than critical interrupt types in items 2–5, noncritical interrupts are immediately followed by the highest priority existing critical interrupt type, without executing any instructions at the noncritical interrupt handler. This is because noncritical interrupt types do not automatically disable MSR mask bits for critical interrupt types (CE and ME). In all other cases, a particular interrupt type listed above automatically disables subsequent interrupts of the same type, as well as all lower priority interrupt types.

5.11 Exception Priorities

Book E requires all synchronous (precise and imprecise) interrupts to be reported in program order, as required by the sequential execution model. The one exception to this rule is the case of multiple synchronous imprecise interrupts. Upon a synchronizing event, all previously executed instructions are required to report any synchronous imprecise interrupt-generating exceptions, and the interrupt is then generated with all of those exception types reported cumulatively in the ESR and in any status registers associated with the particular exception type (such as the FPSCR).

For any single instruction attempting to cause multiple exceptions for which the corresponding synchronous interrupt types are enabled, this section defines the priority order by which the instruction is permitted to cause a single enabled exception, thus generating a particular synchronous interrupt. Note that it is this exception priority

mechanism, along with the requirement that synchronous interrupts be generated in program order, that guarantees that at any given time there exists for consideration only one of the synchronous interrupt types listed in item 1 of Section 5.10.2, “Interrupt Order.” The exception priority mechanism also prevents certain debug exceptions from existing in combination with certain other synchronous interrupt-generating exceptions.

This section does not define the permitted setting of multiple exceptions for which the corresponding interrupt types are disabled. The generation of exceptions for which the corresponding interrupt types are disabled has no effect on the generation of other exceptions for which the corresponding interrupt types are enabled. Conversely, if a particular exception for which the corresponding interrupt type is enabled is shown in the following sections to be of a higher priority than another exception, it prevents the setting of that other exception, independent of whether that other exception’s corresponding interrupt type is enabled or disabled.

Except as specifically noted, only one of the exception types listed for a given instruction type is permitted to be generated at any given time.

NOTE

Some exception types may even be mutually exclusive of each other and could otherwise be considered the same priority. In these cases, the exceptions are listed in the order suggested by the sequential execution model.

The priority of the exception types are listed in the following sections ranging from highest to lowest, within each instruction type.

5.11.1 e500 Interrupt Priorities

The following is a prioritized listing of e500 interrupts:

1. HRESET (Note that hard reset is not defined as a true interrupt in Book E, but is included here to show its relationship to the interrupt structure.)
2. Machine check
3. Unconditional debug event
4. Critical input
5. External input
6. Synchronous interrupts (as defined in Book E)
7. Watchdog timer
8. Fixed-interval timer
9. Performance monitor
10. Decrementer

5.11.2 Exception Priorities for Integer Load, Store, and Cache Management Instructions

The following prioritized list of exceptions may occur as a result of the attempted execution of any integer load, store, and cache management instruction.

1. Debug (instruction address compare)
2. Instruction TLB error (all types)
3. Instruction storage interrupt (all types)
4. Program (illegal instruction)
5. Program (privileged instruction) (**dcbi**)
6. Program (unimplemented operation) (Not supported on the e500. An illegal instruction exception is taken.)
7. Data TLB error (all types)
8. Data storage (all types)
9. Alignment
10. Debug (data address compare)
11. Debug (instruction complete)

If an instruction causes both a debug (instruction address compare) and a debug (data address compare), and is not causing an exception listed in items 2–9, both exceptions can be generated and recorded in the DBSR. A single interrupt results.

5.11.3 Exception Priorities for Allocated Load and Store Instructions

The following prioritized list of exceptions may occur as a result of the attempted execution of any allocated load or store instruction.

1. Debug (instruction address compare)
2. Instruction TLB error (all types)
3. Instruction storage interrupt (all types)
4. Priority of these exceptions is implementation-dependent. However, one and only one exception may be presented to the processor.
 - a) Program (illegal instruction)
 - b) Program (privileged instruction)
 - c) Auxiliary processor unavailable (not supported on the e500)
 - d) Program (unimplemented operation) (not supported on the e500)
5. Data TLB error (all types)
6. Data storage (all types)

7. Alignment
8. Debug (data address compare)
9. Debug (instruction complete)

If the instruction is causing both a debug (instruction address compare) and a debug (data address compare), and is not causing any of the exceptions listed in items 2–9, it is permissible for both exceptions to be generated and recorded in the DBSR. A single debug interrupt results.

5.11.4 Exception Priorities for Allocated Instructions (Other)

The following prioritized list of exceptions may occur as a result of the attempted execution of any allocated instruction that is not a load or store.

1. Debug (instruction address compare)
2. Instruction TLB error (all types)
3. Instruction storage interrupt (all types)
4. Program (illegal instruction)
5. Auxiliary processor unavailable (not supported on the e500)
6. Program (privileged instruction)
7. Program (unimplemented operation) (not supported on the e500)
8. Debug (instruction complete)

5.11.5 Exception Priorities for Privileged Instructions

The following prioritized list of exceptions may occur as a result of the attempted execution of any privileged instruction, except **dcbi**, **rfi**, **rfci**, **rfmci**, and allocated instructions.

1. Debug (instruction address compare)
2. Instruction TLB error (all types)
3. Instruction storage interrupt (all types)
4. Program (illegal instruction)
5. Program (privileged instruction)
6. Program (unimplemented operation) (not supported on the e500)
7. Debug (instruction complete)

See Section 5.11.2, “Exception Priorities for Integer Load, Store, and Cache Management Instructions,” Section 5.11.3, “Exception Priorities for Allocated Load and Store Instructions,” and Section 5.11.4, “Exception Priorities for Allocated Instructions (Other).”

For **mtmsr**, **mtspr** (DBCR0, DBCR1, DBCR2), **mtspr** (TCR), and **mtspr** (TSR), if they are not causing debug (instruction address compare) or program (privileged instruction)

exceptions, it is possible that they are simultaneously enabling (through mask bits) multiple existing exceptions (and at the same time possibly causing a debug (instruction complete) exception). When this occurs, interrupts are handled in the order defined by Section 5.10.2, “Interrupt Order.”

5.11.6 Exception Priorities for Trap Instructions

The following prioritized list of exceptions may occur as a result of the attempted execution of a trap instruction.

1. Debug (instruction address compare)
2. Instruction TLB error (all types)
3. Instruction storage interrupt (all types)
4. Program (illegal instruction)
5. Program (unimplemented operation) (not supported on the e500)
6. Debug (trap)
7. Program (trap)
8. Debug (instruction complete)

If the instruction is causing both a debug (instruction address compare) and a debug (trap), and is not causing any of the exceptions listed in items 2–5, it is permissible for both exceptions to be generated and recorded in the DBSR. A single debug interrupt results.

5.11.7 Exception Priorities for System Call Instruction

The following prioritized list of exceptions may occur as a result of the attempted execution of a System Call instruction.

1. Debug (instruction address compare)
2. Instruction TLB error (all types)
3. Instruction storage interrupt (all types)
4. Program (illegal instruction)
5. Program (unimplemented operation) (not supported on the e500)
6. System call
7. Debug (instruction complete)

5.11.8 Exception Priorities for Branch Instructions

The following prioritized list of exceptions may occur as a result of the attempted execution of any branch instruction.

1. Debug (instruction address compare)
2. Instruction TLB error (both types)
3. Instruction storage interrupt (all types)
4. Program (illegal instruction)
5. Program (unimplemented operation) (Not supported on the e500)
6. Debug (instruction complete, branch taken)

If the instruction is causing both a debug (instruction address compare) and a debug (branch taken), and is not causing any of the exceptions listed in items 2-5, it is permissible for both exceptions to be generated and recorded in the DBSR. A single debug interrupt results.

5.11.9 Exception Priorities for Return from Interrupt Instructions

The following prioritized list of exceptions may occur as a result of the attempted execution of an **rfi**, **rfci**, or **rfmci** instruction.

1. Debug (instruction address compare)
2. Instruction TLB error (both types)
3. Instruction storage interrupt (all types)
4. Program (illegal instruction)
5. Program (privileged instruction)
6. Program (unimplemented operation) (Not supported on the e500)
7. Debug (return from interrupt) (Branch taken is higher priority than instruction complete.)

If **rfi** or **rfci** is causing both a debug (instruction address compare) and a debug (return from interrupt) and is not causing any of the exceptions listed in items 2–5, both exceptions can be generated and recorded in the DBSR. A single debug interrupt results.

5.11.10 Exception Priorities for SPE Instructions

The following list shows the order in which simultaneous exceptions are taken:

1. SPE unavailable exception
2. SPE floating-point data exception
3. SPE floating-point round exception

5.11.11 Exception Priorities for Reserved Instructions

The following prioritized list of exceptions may occur as a result of the attempted execution of any reserved instruction.

1. Debug (instruction address compare)
2. Instruction TLB error (all types)
3. Instruction storage interrupt (all types)
4. Program (illegal instruction)

5.12 Upper Bound on Interrupt Latencies

The core complex interrupt latency is defined here as the number of core clocks between the sampling of the interrupt signal as asserted and the initiation of the IVOR fetch (that is, the fetch of the first instruction in the handler). The interrupt latency of the core complex is determinate unless a guarded load or a cache-inhibited **stwcx.** instruction is being executed. As a guarded load or a cache-inhibited **stwcx.** instruction is being executed, the interrupt latency of the core complex is indeterminate. The minimum interrupt latency for the core complex is 3 core clocks and the maximum is 8. These clocks are described in Table 5-31.

Table 5-31. Interrupt Latency

Action	Guarded Window	Reaching a Recoverable State.	Interrupt Processing	Redirect
Clocks	4/0	2/1	1	1

The interrupt latency defined in Table 5-31 does not include the 2 bus clock cycles required to synchronize the interrupt signal from the pad.

The guarded window is required to ensure that a nonspeculative load/store instruction (such as a **stwcx.** or guarded load) is not issued to the external bus by the load/store unit and then flushed by the completion unit. The guarded window is invoked if and only if a load or store instruction is the oldest instruction being executed (in completion queue entry 0). If an asynchronous interrupt is being serviced and the oldest instruction is not a load/store instruction, the core complex goes straight from sampling the interrupt to ensuring a recoverable state and issuing an exception. If a load/store instruction is oldest, the core complex waits 4 clocks before ensuring a recoverable state. During this time, any instruction finished by the load/store unit is deallocated.

NOTE

Guarded **lmw** and **stmw** instructions can be interrupted before the instruction is completed and then restarted after the interrupt is serviced.

Chapter 6

Power Management

This chapter describes the power management facilities as they are defined by Book E and implemented in devices that contain the e500 core.

6.1 Overview

Power management is a technique that uses the hardware and software resources of the system to minimize the system's power consumption. The scope of this chapter is limited to the features of the core complex only. Additional power management capabilities associated with an integrated device are documented separately.

A complete power management scheme for a system using the core complex requires the support of external logic. The core complex provides software with a mechanism for signaling the need for power management to external logic. It also provides a signal interface that this external logic can use to transition the core complex into its different power management states.

6.2 Power Management States

The core complex operates in one of three power-consumption states: full on, halted, or power down, as shown in Figure 6-1.

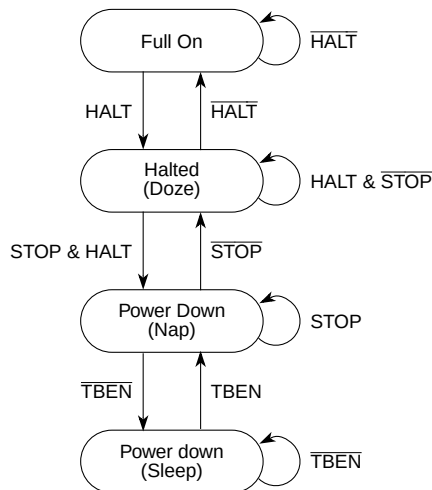


Figure 6-1. Power Management State Diagram

6.2.1 Full On

The full-on state is the default state for the core complex in which all of the internal units are operating at full clock speed. In this state, the core complex reduces unnecessary power dissipation by implementing dynamic power management, in which individual internal functional units may stop clocking automatically whenever they are idle.

6.2.2 Halted

In the halted state, instruction execution is suspended, but clocks continue running in the core complex, and bus snooping continues to support L1 cache coherency. This state is initiated by asserting the HALT input signal to the core complex. The core complex responds by achieving an idle (no instruction execute) state for its internal functional units and CCB interface, and then it asserts the HALTED output to indicate that it is in the halted state.

6.2.3 Power Down

In the power-down state, all internal functional units of the core complex are stopped, and most internal clock distribution is suspended. Clock generation by the internal phase locked loop (PLL) unit continues, however, to allow quick recovery to the halted or full on state. This state is entered by asserting the STOP input to the core complex while it is in the halted state. The core complex responds by turning off the clock distribution to most of its internal functional units (after the CCB interface idles), and then it asserts the STOPPED output to

indicate it is in the power down state. Whether the timebase unit is operating or also stopped while in this state is controlled by the TBEN input.

6.2.3.1 Disabling the Timer

Slight additional power reduction can be achieved in the power-down state by suspending operation of the timer. This is accomplished by negating the timebase enable (TBEN) input signal to the core complex. Note that the TBEN input controls the time base in all power management states. Timer operation is independent of power management except for any special software considerations required for processing timer interrupts that may occur during a power-down state.

Whenever the timer is stopped, software ordinarily uses an external time reference to update the timer state upon restart.

6.2.3.2 Further Power Reduction

While in the power-down state with the timer stopped, further power savings may be achieved for the core complex by stopping the internal PLL clock unit, and optionally the PLL_CLK input. This is done externally by the system after the core complex is safely in the power-down state with the time base off.

On recovering from this complete shutdown state, the system must first restart the PLL and PLL_CLK and allow sufficient time for the PLL to relock before any external interrupt is signaled to the core complex. Because of the long start-up and relock time of the PLL, this complete shutdown state is not suitable for dynamic snooping.

Note that the PLL clock unit is started or stopped by the system through the PLL_CFG[0:5] inputs to the core complex, not through the PLL_CLK input. Simply stopping the PLL_CLK input without changing the PLL_CFG[0:5] inputs causes the PLL unit to lose its lock and clock at varying frequencies as it searches for a lock.

6.3 Interrupts and Power Management

The core complex does not recognize or respond to interrupts when it is in either the halted or power down state. External power management logic must monitor all external interrupt requests (as well as the TBINT output of the processor) to determine if interrupts are being requested. Upon sensing an interrupt request, this external logic will ordinarily negate the STOP and HALT signals to the core complex to restore it to the full on state, which then enables the core complex to respond to the interrupt request.

Depending on the type of interrupt being requested, the core complex saves certain portions of the MSR to MCSRR1, CSRR1, or SRR1 on entry to the interrupt handler routine, and restores them to the MSR on return from the interrupt handler routine. The MSR[WE] bit is always among the bits that are saved and restored. MSR[WE] gates the DOZE, NAP, and

SLEEP power management outputs from the core complex; hence, these outputs negate to the external power management logic on entry to the interrupt, and then return to their previous state on return from the interrupt.

This gated operation of the DOZE, NAP, and SLEEP outputs by the MSR[WE] bit has the following implications for the design of power management software and external logic:

- Assuming that a low-power state was entered in response to the assertion of the DOZE, NAP, or SLEEP output signals, the external logic must recognize that these outputs will remain asserted for some time after the core complex is restored to the full-on state due to the normal latency of starting up the internal clock distribution and initiating the interrupt request, and then negate as the interrupt is serviced.
- The return of the DOZE, NAP, and SLEEP signals to their original state as the MSR[WE] bit is restored is a difference from the way in which power management was implemented on earlier devices that implemented the PowerPC architecture. In previous devices, MSR[POW], which enabled power down requests, was cleared when the processor exited a low-power state without being automatically restored as it is in Book E implementations.

6.4 Snooping in the Power Down State

While the core complex is in the power down state, dynamic bus snooping may still be achieved by the system to maintain coherency in its L1 caches. For this operation, the core complex is momentarily restored to the halted state to perform the snoop operation(s), and then returned to the power down state upon completion of the snoop(s).

Dynamic bus snooping is achieved from the power down state by negating the STOP input to the core complex while the HALT input remains asserted. After the core complex negates the STOPPED output, indicating it has exited the power down state, one or more snoop transactions may be recognized on the CCB interface by the core complex. Snoops should only be issued to the core complex for this dynamic operation while the core is in the halted state and both the STOP and STOPPED signals are negated.

A snoop transaction is normally initiated to the core complex by the assertion of the STS input on the snoop address bus of the CCB interface. If no further snoops are required, the STOP input may then be re-asserted to the core complex as early as the second clock cycle after STS was asserted to return the core complex to the power down state. The core complex automatically returns to the power down state after it has completed the snoop operation in its entirety, including after issuing any required snoop response (after SAACK), and after completing any required snoop copyback transaction on the CCB.

6.5 Power Management Signals

Table 6-1 summarizes the power management signals of the core complex.

Table 6-1. Power Management Signals for Core Complex

Signal	I/O	Description
HALT	I	Asserted by system logic to request the core complex to go into the halted state. Negating this signal causes the core complex to transition back into the full on state. Once asserted, HALT must not be negated until after the core complex has entered the halted state (otherwise the negation may not be recognized).
HALTED	O	Asserted when the core complex is in the halted state. It is the indication that it is safe to go into the power down state.
STOP	I	Asserted by system logic to request that the core complex go from the halted state into the power down state. Negating this signal causes the core complex to transition back into the halted state. Once asserted, STOP must not be negated until after the core complex has entered the stopped state (otherwise the negation may not be recognized). For power management purposes, STOP must only be asserted while the core complex is in the halted state.
STOPPED	O	Asserted any time the internal functional clocks of the core complex are stopped.
TBEN	I	Asserted by the system logic to enable the time base.
TBINT	O	Asserted when a time base interrupt is signalled. This will ordinarily prompt external logic to bring the core out of power down mode so that the interrupt can be serviced.
DOZE	O	These signals reflect the state of HID0[DOZE], HID0[NAP], and HID0[SLEEP] bits respectively, further qualified with MSR[WE] = 1 (both must be 1 for the respective output to be asserted). The state of these signals has no effect on the power-down state of the core complex. They serve only as indicators to external logic of power management requests by software.
NAP	O	
SLEEP	O	

6.6 Power Management Control Bits

All power management of the core is initiated by logic outside the core. However, the core provides control bits that are used to signal the power management requests to external logic. These bits are:

- **MSR[WE]**—The WE bit is used to qualify the values of the HID0[DOZE], HID0[NAP], and HID0[SLEEP] bits in the generation of the DOZE, NAP, and SLEEP signals to the system logic.
- **HID0[DOZE]**—Qualified with MSR[WE], signals external logic to initiate doze mode (corresponding to the halted state).
- **HID0[NAP]**—Qualified with MSR[WE], signals external logic to initiate nap mode (corresponding to power down state with the time base enabled).
- **HID0[SLEEP]**—Qualified with MSR[WE], signals external logic to initiate sleep mode (corresponding to the power down state with the time base disabled).

6.7 Software Considerations for Power Management

Setting MSR[WE] generates a request to the system logic (external to the core complex) to enter a power saving state. The power saving state (doze, nap, or sleep) must be previously declared with the appropriate HID0 bit, typically at system start-up time. Setting

Software Considerations for Power Management

MSR[WE] has no direct effect on instruction execution, but it is reflected on the DOZE, NAP, and SLEEP signals, depending on the setting of HID0[DOZE], HID0[NAP], and HID0[SLEEP]. To ensure a clean transition into and out of a power-saving mode, the following program sequence is recommended:

```
        msync  
        mtmsr (WE)  
        isync  
loop:   br loop
```

Chapter 7

Performance Monitor

This chapter describes the performance monitor, which is implemented as an APU on the e500 core. The performance monitor is generally defined by the Motorola Book E implementation standards, although some aspects of the implementation are defined by the e500 implementation. In particular, the events that are counted are specific to the e500 microarchitecture.

7.1 Overview

The e500 core complex architecture defines a performance monitor facility that provides the ability to monitor and count predefined events such as processor clocks, misses in the instruction cache or data cache, types of instructions decoded, or mispredicted branches. The count of such events can be used to trigger the performance monitor exception.

The performance monitor can be used to do the following:

- Improve system performance by monitoring software execution and then recoding algorithms for more efficiency. For example, memory hierarchy behavior can be monitored and analyzed to optimize task scheduling or data distribution algorithms.
- Characterize processors in environments not easily characterized by a benchmark or trace
- Help system developers bring up and debug their systems

The performance monitor uses the following resources:

- The performance monitor mark bit in the MSR (MSR[PMM]). This bit controls which programs are monitored.
- The move to/from PMR instructions, **mtpmr** and **mfpmr**.
- The external input pin, PM_EVENT.
- Performance monitor registers (PMRs):
 - The performance monitor counter registers (PMC0–PMC3) are 32-bit counters used to count software-selectable events. Each counter counts up to 64 reference events and up to 64 counter-specific events. UPMC0–UPMC3 provide user-level read access to these registers. Reference events are those that should be

applicable to most microprocessor microarchitectures and be of general value. They are identified in Table 7-8.

- The performance monitor global control register (PMGC0) controls the counting of performance monitor events. It takes priority over all other performance monitor control registers. UPMGC0 provides user-level read access to this register.
- The performance monitor local control registers (PMLCa0–PMLCa3, PMLCb0–PMLCb3) control each individual performance monitor counter. Each counter has a corresponding PMLCa and PMLCb register. UPMLCa0–UPMLCa3 and UPMLCb0–UPMLCb3 provide user-level read access to PMLCa0–PMLCa3, PMLCb0–PMLCb3).
- The performance monitor interrupt follows the Book E interrupt model and is assigned to interrupt vector offset register 35 (IVOR35). Its priority is less than the fixed-interval interrupt and greater than the decremter interrupt.

The e500 core complex implements a performance monitor as an APU. Software communication with the performance monitor APU is achieved through performance monitor registers (PMRs) rather than SPRs. The PMRs are used for enabling conditions that can trigger a APU-defined performance monitor interrupt.

7.2 Performance Monitor APU Programming Model

The performance monitor APU provides a set of PMRs for defining, enabling, and counting conditions that trigger the performance interrupt. The APU defines instructions for reading and writing the PMRs.

7.2.1 Performance Monitor APU Registers

The performance monitor APU defines IVOR35 (SPR 531) for indicating the address of the performance monitor interrupt vector. IVOR35 is described in Section 2.7.1.7, “Interrupt Vector Offset Registers (IVORs).”

The supervisor-level performance monitor registers in Table 7-1 are accessed with **mtpmr** and **mfpmr**. Attempting to read or write supervisor-level registers while in user-mode causes a privilege exception.

Table 7-1. Performance Monitor Registers—Supervisor Level

Number	pmr[0–4]	pmr[5–9]	Name	Symbol
16	00000	10000	Counter 0	PMC0
17	00000	10001	Counter 1	PMC1
18	00000	10010	Counter 2	PMC2
19	00000	10011	Counter 3	PMC3

Table 7-1. Performance Monitor Registers—Supervisor Level (continued)

Number	pmr[0–4]	pmr[5–9]	Name	Symbol
144	00100	10000	Local control a0	PMLCa0
145	00100	10001	Local control a1	PMLCa1
146	00100	10010	Local control a2	PMLCa2
147	00100	10011	Local control a3	PMLCa3
272	01000	10000	Local control b0	PMLCb0
273	01000	10001	Local control b1	PMLCb1
274	01000	10010	Local control b2	PMLCb2
275	01000	10011	Local control b3	PMLCb3
400	01100	10000	Global control 0	PMGC0

The user-level performance monitor registers in Table 7-2 are read-only and are accessed with the **mfpmr** instruction. Attempting to write user-level registers in either supervisor or user mode causes an illegal instruction exception.

Table 7-2. Performance Monitor Registers—User Level (Read-Only)

Number	pmr[0–4]	pmr[5–9]	Name	Symbol
0	00000	00000	Counter 0	UPMC0
1	00000	00001	Counter 1	UPMC1
2	00000	00010	Counter 2	UPMC2
3	00000	00011	Counter 3	UPMC3
128	00100	00000	Local control a0	UPMLCa0
129	00100	00001	Local control a1	UPMLCa1
130	00100	00010	Local control a2	UPMLCa2
131	00100	00011	Local control a3	UPMLCa3
256	01000	00000	Local control b0	UPMLCb0
257	01000	00001	Local control b1	UPMLCb1
258	01000	00010	Local control b2	UPMLCb2
259	01000	00011	Local control b3	UPMLCb3
384	01100	00000	Global control 0	UPMGC0

7.2.1.1 Global Control Register 0 (PMGC0)

The performance monitor global control register (PMGC0), shown in Figure 7-1, controls all performance monitor counters.

	32	33	34	35	63
Field	FAC	PMIE	FCECE	—	
Reset	All zeros				
R/W	PMGC0: Supervisor R/W; UPMGC0: User read-only				
PMR	PMGC0: PMR400; UPMGC0:PMR384				

**Figure 7-1. Performance Monitor Global Control Register 0 (PMGC0)
User Performance Monitor Global Control Register 0 (UPMGC0)**

PMGC0 is cleared by a hard reset. Reading this register does not change its contents. Table 7-3 describes PMGC0 fields.

Table 7-3. PMGC0 Field Descriptions

Bits	Name	Description
32	FAC	Freeze all counters. When FAC is set by hardware or software, PMLCx[FC] maintains its current value until changed by software. 0 The PMCs are incremented (if permitted by other performance monitor control bits). 1 The PMCs are not incremented.
33	PMIE	Performance monitor interrupt enable. Enabled conditions and events are described in Section 7.4. 0 Performance monitor interrupts are disabled. 1 Performance monitor interrupts are enabled and occur when an enabled condition or event occurs.
34	FCECE	Freeze counters on enabled condition or event. Enabled conditions and events are described in Section 7.4. 0 The PMCs can be incremented (if permitted by other performance monitor control bits). 1 The PMCs can be incremented (if permitted by other performance monitor control bits) only until an enabled condition or event occurs. When an enabled condition or event occurs, PMGC0[FAC] is set. It is the user's responsibility to clear FAC.
35–63	—	Reserved, should be cleared.

7.2.1.2 User Global Control Register 0 (UPMGC0)

The contents of PMGC0 are reflected to UPMGC0, which can be read by user-level software. UPMGC0 can be read with the **mfpmr** instruction using PMR384.

7.2.1.3 Local Control A Registers (PMLCa0–PMLCa3)

The local control A registers 0 through 3 (PMLCa0–PMLCa3) function as event selectors and give local control for the corresponding performance monitor counters. PMLCa works with the corresponding PMLCb register. PMLCa registers are shown in Figure 7-2.

	32	33	34	35	36	37	38	40	41	47	48	63
Field	FC	FCS	FCU	FCM1	FCM0	CE	—	EVENT				
Reset	All zeros											
R/W	PMLCa0–PMLCa3: Supervisor R/W UPMLCa0–UPMLCa3: User read-only											
PMR	PMLCa0: PMR144, PMLCa1: PMR145, PMLCa2.: PMR146, PMLCa3: PMR147 UPMLCa0: PMR128, UPMLCa1: PMR129, UPMLCa2.: PMR130, UPMLCa3: PMR131											

**Figure 7-2. Local Control A Registers (PMLCa0–PMLCa3)
User Local Control A Registers (UPMLCa0–UPMLCa3)**

Table 7-4 describes PMLCa fields.

Table 7-4. PMLCa0–PMLCa3 Field Descriptions

Bits	Name	Description
32	FC	Freeze counter. 0 The PMC is incremented (if permitted by other performance monitor control bits). 1 The PMC is not incremented.
33	FCS	Freeze counter in supervisor state. 0 The PMC is incremented (if permitted by other performance monitor control bits). 1 The PMC is not incremented if MSR[PR] is cleared.
34	FCU	Freeze counter in user state. 0 The PMC is incremented (if permitted by other performance monitor control bits). 1 The PMC is not incremented if MSR[PR] is set.
35	FCM1	Freeze counter while mark is set. 0 The PMC is incremented (if permitted by other performance monitor control bits). 1 The PMC is not incremented if MSR[PMM] is set.
36	FCM0	Freeze counter while mark is cleared. 0 The PMC is incremented (if permitted by other performance monitor control bits). 1 The PMC is not incremented if MSR[PMM] is cleared.
37	CE	Condition enable. 0 Overflow conditions for PMCx cannot occur (PMCx cannot cause interrupts, cannot freeze counters) 1 Overflow conditions occur when the most-significant-bit of PMCx is equal to 1. It is recommended that CE be cleared when counter PMCx is selected for chaining.
38–40	—	Reserved, should be cleared.
41–47	EVENT	Event selector. Up to 128 events selectable. See Section 7.7, “Event Selection”
48–63	—	Reserved, should be cleared.

7.2.1.4 User Local Control A Registers (UPMLCa0–UPMLCa3)

The contents of PMLCa0–PMLCa3 are reflected to UPMLCa0–UPMLCa3, which can be read by user-level software with **mfpmr** using PMR numbers in Table 7-2.

7.2.1.5 Local Control B Registers (PMLCb0–PMLCb3)

Local control B registers 0 through 3 (PMLCb0–PMLCb3) specify a threshold value and a multiple to apply to a threshold event selected for the corresponding performance monitor counter. For the e500, thresholding is supported only for PMC0 and PMC1. PMLCb works with the corresponding PMLCa register. PMLCb registers are shown in Figure 7-3.

	32	52	53	55	56	57	58	63
Field	—			THRESHMUL	—		THRESHOLD	
Reset	All zeros							
R/W	PMLCb0–PMLCb3: Supervisor R/W UPMLCb0–UPMLCb3: User read-only							
PMR	PMLCb0: PMR272, PMLCb1: PMR273, PMLCb2.: PMR274, PMLCb3: PMR275 UPMLCb0: PMR256, UPMLCb1: PMR257, UPMLCb2.: PMR258, UPMLCb3: PMR259							

**Figure 7-3. Local Control B Registers (PMLCb0–PMLCb3)
User Local Control B Registers (UPMLCb0–UPMLCb3)**

Table 7-5 describes PMLCb fields.

Table 7-5. PMLCb0–PMLCb3 Field Descriptions

Bits	Name	Description
32–52	—	Reserved, should be cleared.
53–55	THRESHMUL	Threshold multiple. 000 Threshold field is multiplied by 1 (PMLCbn[THRESHOLD] * 1) 001 Threshold field is multiplied by 2 (PMLCbn[THRESHOLD] * 2) 010 Threshold field is multiplied by 4 (PMLCbn[THRESHOLD] * 4) 011 Threshold field is multiplied by 8 (PMLCbn[THRESHOLD] * 8) 100 Threshold field is multiplied by 16 (PMLCbn[THRESHOLD] * 16) 101 Threshold field is multiplied by 32 (PMLCbn[THRESHOLD] * 32) 110 Threshold field is multiplied by 64 (PMLCbn[THRESHOLD] * 64) 111 Threshold field is multiplied by 128 (PMLCbn[THRESHOLD] * 128)
56–57	—	Reserved, should be cleared.
58–63	THRESHOLD	Threshold. Only events that exceed the THRESHOLD value are counted. Events to which a threshold value applies are implementation-dependent as are the dimension (for example duration in cycles) and the granularity with which the threshold value is interpreted. By varying the threshold value, software can obtain a profile of the event characteristics subject to thresholding. For example, if PMC1 is configured to count cache misses that last longer than the threshold value, software can obtain the distribution of cache miss durations for a given program by monitoring the program repeatedly using a different threshold value each time.

7.2.1.6 User Local Control B Registers (UPMLCb0–UPMLCb3)

The contents of PMLCb0–PMLCb3 are reflected to UPMLCb0–UPMLCb3, which can be read by user-level software with **mfpmr** using PMR numbers in Table 7-2.

7.2.1.7 Performance Monitor Counter Registers (PMC0–PMC3)

The performance monitor counter registers (PMC0–PMC3), shown in Figure 7-4, are 32-bit counters that can be programmed to generate interrupt signals when they overflow. Each counter is enabled to count 128 events.

	32	33	63
Field	OV	Counter value	
Reset	All zeros		
R/W	PMC0–PMC3: Supervisor R/W UPMC0–UPMC3: User read-only		
PMR	PMC0: PMR16, PMC1: PMR17, PMC2,: PMR18, PMC3: PMR19 UPMC0: PMR0, UPMC1: PMR1, UPMC2,: PMR2, UPMC3: PMR3		

**Figure 7-4. Performance Monitor Counter Registers (PMC0–PMC3)
User Performance Monitor Counter Registers (UPMC0–UPMC3)**

Table 7-6 describes PMC register fields.

Table 7-6. PMC0–PMC3 Field Descriptions

Bits	Name	Description
32	OV	Overflow. When this bit is set, it indicates this counter has reached its maximum value.
33–63	Counter Value	Indicates the number of occurrences of the specified event.

Counters overflow when the high-order bit (the sign bit) becomes set; that is, they reach the value 2,147,483,648 (0x8000_0000). However, an exception is not signaled unless PMGC0[PMIE] and PMLCax[CE] are also set appropriately.

Note that the interrupts are masked by clearing MSR[EE]. An exception may be signaled while MSR[EE] is zero, but the interrupt is not taken until MSR[EE] is set. Setting PMGC0[FCECE] forces counters to stop counting when an enabled condition or event occurs.

Software is expected to use **mtpmr** to explicitly set PMCs to non-overflowed values. Setting an overflowed value may cause an erroneous exception. For example, if both PMGC0[PMIE] and PMLCax[CE] are set and an **mtpmr** loads an overflowed value into PMCx, an interrupt may be generated without an event counting ever occurring.

7.2.1.8 User Performance Monitor Counter Registers (UPMC0–UPMC3)

The contents of PMC0–PMC3 are reflected to UPMC0–UPMC3, which can be read by user-level software with the **mfpmr** instruction using PMR numbers in Table 7-2..

7.3 Performance Monitor APU Instructions

The APU also defines the instructions in Table 7-7 to move to and move from these PMRs. Full descriptions of these instructions can be found in the EREF.

Table 7-7. Performance Monitor APU Instructions

Name	Mnemonic	Syntax
Move from Performance Monitor Register	mfpmr	rD,PMRN
Move to Performance Monitor Register	mtpmr	PMRN,rS

7.4 Performance Monitor Interrupt

The performance monitor interrupt is triggered by an enabled condition or event such as the following:

- A PMCx register overflow condition occurs with the following settings:
 - the counter's overflow condition is enabled; PMLCax[CE] is set
 - the counter indicates an overflow; PMCx[32] is set

PMGC0[PMIE] must be set for a performance monitor exception to be signaled on an enabled condition or event.

When PMGC0[FCECE] is set the performance monitor can freeze counters triggered by an enabled condition or event.

Although the exception condition could occur with MSR[EE] cleared, the interrupt cannot be taken until MSR[EE] is set. If a counter overflows while PMGC0[FCECE] is cleared, PMLCa[CE] is set, and MSR[EE] is cleared, it is possible for the counter to wrap around to all zeros again without the performance monitor interrupt being taken.

7.5 Event Counting

This section describes configurability and specific unconditional counting modes.

7.5.1 Processor Context Configurability

Counting can be enabled if conditions in the processor state match a software-specified condition. Because a software task scheduler may switch a processor's execution among multiple processes and because statistics on only a particular process may be of interest, a facility is provided to mark a process. The performance monitor mark bit, MSR[PMM], is used for this purpose. System software may set this bit when a marked process is running. This enables statistics to be gathered only during the execution of the marked process. The states of MSR[PR] and MSR[PMM] together define a state that the processor (supervisor or user) and the process (marked or unmarked) may be in at any time. If this state matches

an individual state specified in the PMLCax, the state for which monitoring is enabled, counting is enabled.

The following are states that can be monitored:

- (Supervisor) only
- (User) only
- (Marked and user) only
- (Not marked and user) only
- (Marked and supervisor) only
- (Not marked and supervisor) only
- (Marked) only
- (Not marked) only

In addition, one of two unconditional counting modes may be specified:

- Counting is unconditionally enabled regardless of the states of MSR[PMM] and MSR[PR]. This can be accomplished by clearing PMLCax[FCS], PMLCax[FCU], PMLCax[FCM1], and PMLCax[FCM0] for each counter control.
- Counting is unconditionally disabled regardless of the states of MSR[PMM] and MSR[PR]. This is done by setting PMGC0[FAC] or PMLCax[FC].

7.6 Example Usage

The following sections provide examples of how to use the performance monitor facility:

7.6.1 Chaining Counters

In the event that a user would like to decrease cache contamination due to performance monitor interrupts, the counter chaining feature enables a higher event count than is possible with a single counter. Chaining two counters together effectively adds 32 bits to a counter register. By setting the event to count the number of PMC carryout generations of another counter, the chained counter is incremented each time that PMCx register count rolls over to zero. Multiple counters may be chained together.

Since it is not possible to read the entire chained value in a single instruction, a carry may occur between counter reads, resulting in an inaccurate value. A sequence like the following is necessary to read the complete chained value when it spans multiple counters. The example shown is for a two-counter case.

```
loop:  mfpmr          Rx,pmctr1      #load from upper counter
      mfpmr          Ry,pmctr0      #load from lower counter
      mfpmr          Rz,pmctr1      #load from upper counter
      cmp            cr0,0,Rz,Rx    #see if 'old' = 'new'
      bc             4,2,loop       #loop if carry occurred between reads
```

The comparison and loop are necessary to ensure that a consistent set of values has been obtained.

7.6.2 Thresholding

Threshold event measurement enables the counting of duration and usage events. For example, dLFB load miss cycles (event #) requires a threshold value. A dLFB load miss cycles event is counted only when the number of cycles spent recovering from the miss is greater than the threshold. Because this event is a reference event and every counter has an individual threshold, one execution of a performance monitor program can sample two different threshold values. Measuring code performance with multiple concurrent thresholds expedites code profiling significantly.

7.7 Event Selection

Event selection is specified through the PMLCax registers described in Table 7-4. The event-select fields in PMLCax[EVENT] are described Table 7-8, which lists encodings for the selectable events to be monitored. Table 7-8 establishes a correlation between each counter, events to be traced, and the pattern required for the desired selection.

Note the following with respect to Table 7-8.

- The number column indicates the counters and event selects assigned to a given event.
 - Reference events, which should be applicable to most microprocessors, are labelled Ref:#. Reference events are shared across counters PMC0 through PMC3.
 - Common events, which are fairly specific to the e500 microarchitecture are labelled Com:#. Like reference events, common events are shared across counters PMC0–PMC3.
 - Counter specific events, which are counted only on one or more specific counters, are labelled C[0–3]:#. This notation indicates the counter to which the event is assigned. For example, a specific event assigned to counter PMC2 is shown as C2:#.
- The perfect column indicates whether the event count includes any occurrences due to processing that was not architecturally required by the PowerPC sequential execution model (speculative processing).
 - No indicates that some counts occurred because of instructions that were later flushed, and the counter contains only an approximation of the desired value.
 - Yes means the counter provides an exact count uncontaminated by any speculative processing.

- For load/store events, a micro-op is described as translated when the micro-op has successfully translated and is in the second stage of the load/store translate pipeline.
- For chaining events, if a counter is configured to count its own carry-out bit, that counter does not increment. For example, if PMC2 is selected to count PMC2 carry-out events, then PMC2 does not increment.

Table 7-8. Performance Monitor Event Selection

Number	Event	Perfect	Description
General Events			
Ref:0	Nothing	Yes	Register counter will hold current value
Ref:1	Processor cycles	Yes	Count every processor cycle
Ref:2	Instructions completed	Yes	Counts completed instructions. 0, 1, or 2 per cycle.
Com:3	Micro-ops completed	Yes	Counts completed micro-ops. 0, 1, or 2 per cycle. (Counts 2 for load/store-with-update)
Com:4	Instructions fetched	No	Counts fetched instructions. 0, 1, 2, 3, or 4 per cycle. (Number of instructions written to the instruction buffer.)
Com:5	Micro-ops decoded	No	Counts micro-ops decoded. 0, 1, or 2 per cycle. (Counts 2 for load/store-with-update)
Com:6	PM_EVENT transitions	No	This counts the number of 0 to 1 transitions on the PM_EVENT input pin.
Com:7	PM_EVENT cycles	No	This counts the number of processor cycles which occur when the PM_EVENT input pin is 1.
Instruction Types Completed			
Com:8	Branch instructions completed	Yes	Counts completed branch instructions. (b*)
Com:9	Load micro-ops completed	Yes	Counts completed load micro-ops. (l*, evl*, load-update, load-multiple, load-context, dcbl(L1), and dcblst(L1))
Com:10	Store micro-ops completed	Yes	Counts completed store micro-ops. (st*, evst*, store-update, store-multiple, tlblvax, icbi, icblc, icblts, dcba, dcbf, dcbic, dcbst, dcbt(L2), dcbtts, dcbtst(L2), dcbtstts, dcbz, icbt(L2), mbar, and msync)
Com:11	Number of completion buffer redirects	Yes	
Branch Prediction and Execution Events			
Com:12	Branches finished	No	
Com:13	Taken branches finished	No	
Com:14	Bifed branches finished	No	
Com:15	Branches mispredicted	No	Counts branch instructions mispredicted due to direction, target, or IAB prediction. Does not count phantom branches.
Com:16	Branches mispredicted due to direction prediction	No	Counts branch instructions mispredicted due to direction prediction.

Table 7-8. Performance Monitor Event Selection (continued)

Number	Event	Perfect	Description
Com:17	BTB hits and pseudo-hits	No	Counts branch instructions that either hit in the BTB, or missed in the BTB but are not-taken (a "pseudo-hit"). Used to compute upper bound on prediction rate.
Pipeline Stalls			
Com:18	Cycles decode stalled	No	Cycles the instruction buffer is not empty but 0 instructions decoded
Com:19	Cycles issue stalled	No	Cycles the issue buffer is not empty but 0 instructions issued
Com:20	Cycles branch issue stalled	No	Cycles the branch buffer is not empty but 0 instructions issued
Com:21	Cycles SRS0 schedule stalled	No	Cycles SRS0 is not empty but 0 instructions scheduled
Com:22	Cycles SRS1 schedule stalled	No	Cycles SRS1 is not empty but 0 instructions scheduled
Com:23	Cycles VRS schedule stalled	No	Cycles VRS is not empty but 0 instructions scheduled
Com:24	Cycles LRS schedule stalled	No	Cycles LRS is not empty but 0 instructions scheduled
Com:25	Cycles BRS schedule stalled	No	Cycles BRS is not empty but 0 instructions scheduled
Load/Store, Data Cache, and dLFB Events			
Com:26	Total translated	No	Total Ldst microops translated.
Com:27	Loads translated	No	Number of cacheable L* or EVL* microops translated. (This includes microops from load-multiple, load-update, and load-context instructions.)
Com:28	Stores translated	No	Number of cacheable ST* or EVST* microops translated. (This includes microops from store-multiple, store-update, and save-context instructions.)
Com:29	Touches translated	No	Number of cacheable DCBT and DCBTST instructions translated (L1 only) (Doesn't count touches that are converted to nops i.e. exceptions, noncacheable, hid0[noptij] bit is set.)
Com:30	Cacheops translated	No	Number of dcba , dcbf , dcbst , and dcbz instructions translated (e500 traps on dcbi)
Com:31	Cache-inhibited accesses translated	No	Number of cache inhibited accesses translated
Com:32	Guarded loads translated	No	Number of guarded loads translated
Com:33	Write-through stores translated	No	Number of write-through stores translated
Com:34	Misaligned load or store accesses translated	No	Number of misaligned load or store accesses translated.
Com:35	Total allocated to dLFB	No	
Com:36	Loads translated and allocated to dLFB	No	(Applies to same class of instructions as loads translated.)

Table 7-8. Performance Monitor Event Selection (continued)

Number	Event	Perfect	Description
Com:37	Stores completed and allocated to dLFB	Yes	(Applies to same class of instructions as stores translated.)
Com:38	Touches translated and allocated to dLFB	No	(Applies to same class of instructions as touches translated.)
Com:39	Stores completed	Yes	Number of cacheable ST* or EVST* microops completed. (Applies to the same class of instructions as stores translated.)
Com:40	dL1 locks	Yes	Number of cache lines locked in the dL1. (Will count a lock even if an overlock condition is encountered.)
Com:41	dL1 reloads	No	This is historically used to determine dcache miss rate (along with loads/stores completed). This counts dL1 reloads for any reason.
Com:42	dL1 castouts	No	Does not count castouts due to DCBF.
Com:43	Times detected replay condition: Load miss with dLFB full.	No	
Com:44	Times detected replay condition: Load miss with load queue full.	No	
Com:45	Times detected replay condition: Load guarded miss when the load is not yet at the bottom of the completion buffer.	No	
Com:46	Times detected replay condition: Translate a store when the StQ is full.	No	
Com:47	Times detected replay condition: Address collision.	No	
Com:48	Times detected replay condition: DMMU miss.	No	
Com:49	Times detected replay condition: DMMU busy.	No	
Com:50	Times detected replay condition: Second part of misaligned access when first part missed in cache.	No	
Com:51	Cycles stalled on replay condition: Load miss with dLFB full.	No	
Com:52	Cycles stalled on replay condition: Load miss with load queue full.	No	

Table 7-8. Performance Monitor Event Selection (continued)

Number	Event	Perfect	Description
Com:53	Cycles stalled on replay condition: Load guarded miss when the load is not yet at the bottom of the completion buffer.	No	
Com:54	Cycles stalled on replay condition: Translate a store when the StQ is full.	No	
Com:55	Cycles stalled on replay condition: Address collision.	No	
Com:56	Cycles stalled on replay condition: DMMU miss.	No	
Com:57	Cycles stalled on replay condition: DMMU busy.	No	
Com:58	Cycles stalled on replay condition: Second part of misaligned access when first part missed in cache.	No	
Fetch, Instruction Cache, iLFB, and iPrefetch Events			
Com:59	iL1 locks	Yes	Number of cache lines locked in the iL1. (Will count a lock even if an overlock condition is encountered.)
Com:60	iL1 reloads from fetch	No	This is historically used to determine icache miss rate (along with instructions completed) Reloads due to demand fetch.
Com:61	Number of fetches	No	Counts the number of fetches that write at least one instruction to the instruction buffer. (With instruction fetched, can used to compute instructions-per-fetch)
iMMU, dMMU and L2MMU Events			
Com:62	iMMU TLB4K reloads	No	
Com:63	iMMU VSP reloads	No	
Com:64	dMMU TLB4K reloads	No	
Com:65	dMMU VSP reloads	No	
Com:66	L2MMU misses	Yes	Counts iTLB/dTLB error interrupt.
BIU Interface Usage			
Com:67	BIU master requests	No	Number of master transactions. (Number of master TSs.)
Com:68	BIU master I-Side requests	No	Number of master I-Side transactions. (Number of master I-Side TSs.)
Com:69	BIU master D-Side requests	No	Number of master D-Side transactions. (Number of master D-Side TSs.)

Table 7-8. Performance Monitor Event Selection (continued)

Number	Event	Perfect	Description
Com:70	BIU master D-side castout requests	No	Number of master D-Side non-program-demand castout transactions. This counts replacement pushes and snoop pushes. This does not count DCBF castouts. (Number of master D-side non-program-demand castout TSSs.)
Com:71	BIU master retries	No	Number of transactions which were initiated by this processor which were retried on the BIU interface. (Number of master ARTRYs.)
Snoop			
Com:72	Snoop requests	N/a	Number of externally generated snoop requests. (Counts snoop TSSs.)
Com:73	Snoop hits	N/a	Number of snoop hits on all D-side resources regardless of the cache state (modified, exclusive, or shared)
Com:74	Snoop pushes	N/a	Number of snoop pushes from all D-side resources. (Counts snoop ARTRY/WOPs.)
Com:75	Snoop retries	N/a	Number of snoop requests retried. (Counts snoop ARTRYs.)
Threshold Events			
C0:76 C1:76	dLFB load miss cycles	No	This threshold event counts the number of times the number of cycles between a load allocation in the dLFB and write-back to the dL1 exceeds the threshold value. This is done on one fixed entry (entry 0) in the dLFB.
C0:77 C1:77	iLFB fetch miss cycles	No	This threshold event counts the number of times the number of cycles between allocation in the iLFB and write-back to the iL1 exceeds the threshold value. This is done on one fixed entry (entry 0) in the iLFB.
C0:78 C1:78	External input interrupt latency cycles	N/a	This threshold event counts the number of times the number of cycles between pin request for interrupt asserted (but possibly masked/disabled) and redirecting fetch to the external interrupt vector exceeds the threshold value.
C0:79 C1:79	Critical input interrupt latency cycles	N/a	This threshold event counts the number of times the number of cycles between pin request for critical interrupt asserted (but possibly masked/disabled) and redirecting fetch to the critical interrupt vector exceeds the threshold value.
C0:80 C1:80	External input interrupt pending latency cycles	N/a	This threshold event counts the number of times the number of cycles between external interrupt pending (enabled and pin asserted) and redirecting fetch to the external interrupt vector exceeds the threshold value. Note that this and the next event may count multiple times for a single interrupt if the threshold is very small, and the interrupt is masked a few cycles after it is asserted, and eventually becomes unmasked.
C0:81 C1:81	Critical input interrupt pending latency cycles	N/a	This threshold event counts the number of times the number of cycles between pin request for critical interrupt pending (enabled and pin asserted) and redirecting fetch to the critical interrupt vector exceeds the threshold value. See note for previous event.
Chaining Events			
Com:82	PMC0 carry-out	N/a	Counts the number of times PMC0[32] transitioned from 1 to 0.

Table 7-8. Performance Monitor Event Selection (continued)

Number	Event	Perfect	Description
Com:83	PMC1 carry-out	N/a	Counts the number of times PMC1[32] transitioned from 1 to 0.
Com:84	PMC2 carry-out	N/a	Counts the number of times PMC2[32] transitioned from 1 to 0.
Com:85	PMC3 carry-out	N/a	Counts the number of times PMC3[32] transitioned from 1 to 0.
Interrupt Events			
Com:86	Number of interrupts taken	Yes	
Com:87	Number of external input interrupts taken	Yes	
Com:88	Number of critical input interrupts taken	Yes	
Com:89	Number of system call and trap interrupts	Yes	

Chapter 8

Debug Support

The scope of the following discussion is limited to the features of the core complex only. Additional debug capabilities associated with an integrated device are documented elsewhere.

8.1 Overview

Internal debug mechanisms allow for software and hardware debug by providing debug functions, such as instruction and data breakpoints and program trace mode. The debug facilities consist of a set of software-accessible debug registers and interrupt mechanisms.

Access to the internal debug mode facilities using the **mfsprr** and **mtsprr** instructions, as well as the debug interrupt mechanism, are defined as part of Book E. Reference that document for more information.

8.1.1 Deviations from Book E Debug Mode

The e500 core complex supports Book E debug mode with the following exceptions:

Instruction address compare 3 and 4, (IAC3, IAC4), and data address compare 3 and 4, (DAC3, DAC4), along with their debug exceptions are not implemented. All address comparisons are based on effective addresses only.

- Data value compare (DVC) and its debug exception is not supported.
- IAC3, IAC4, DAC3, DAC4, and data value compare (DVC), debug exceptions are not supported.
- Only effective addresses are compared with instruction address compare (IAC1, IAC2), and data address compare (DAC1, DAC2).
- Debug RET events for the **rfci** instruction are not logged if MSR[DE] is cleared.

Table 8-1 describes the reset field in debug control register 0, (DBCR0[RST]).

Table 8-1. DBCR0[RST] Field Description

Bit	Name	Description
34–35	RST	0x Default. 1x Causes a hard reset if MSR[DE] and DBCR0[IDM] are set. Always cleared on subsequent cycle.

Table 8-2 describes the most recent reset (MRR) field in the debug status register (DBSR).

Table 8-2. DBSR[MRR] Field Description

Bit	Name	Description
34–35	MRR	Most recent reset. Set of three values when a reset occurs. Undefined at power-up. The e500 implements MRR as follows: 0x No hard reset occurred since this bit was last cleared by software. 1x The previous reset was a hard reset.

8.1.2 Software Facilities

The core complex implements a subset of the Book E internal debug features. The following facilities are not implemented:

- Instruction address compare 3 and 4 are not implemented. Instruction address compare is only based on effective addresses.
- Data address compare 1 and 2 only compare effective addresses.
- Data value compare (DVC registers) is not supported.

8.1.3 Hardware Facilities

is a it does not contain IEEE 1149.1 standard boundary cells on its I/O periphery, it should not be considered IEEE 1149.compliant.

8.2 TAP Controller and Register Model

JTAG (joint test action group) is a serial protocol that specifies data flow through special registers connected between test data in (TDI) and test data out (TDO). Figure 8-1 shows the TAP registers implemented by the core complex. For more information, refer to *IEEE Standard Test Access Port and Boundary Scan Architecture IEEE STD 1149-1a-1993*.

Figure 8-1. TAP Controller with Supported Registers

8.2.1 TAP Interface Signals

The TAP interface signals are summarized in Table 8-3 and discussed briefly in the following sections. The test data input (TDI) and test data output (TDO) scan ports are used to scan instructions and data into the various scan registers for JTAG operations. The scan

operation is controlled by the TAP controller, which in turn is controlled by the test mode select (TMS) input sequence. The scan data is latched at the rising edge of test clock (TCK).

The TAP and boundary-scan logic are not used under typical operating conditions. Detailed discussion of all e500 test functions is beyond the scope of this document. However, sufficient information is provided to allow the system designer to disable test functions that would impede normal operation.

Table 8-3. TAP/IEEE/JTAG Interface Signal Summary

Signal Name	Description	Input/Output	IEEE 1149.1a Function
TCK	Test clock	In	Scan clock
TDI	Test data input	In	Serial scan input signal
TDO	Test data output	Out	Serial scan output signal
TMS	Test mode select	In	TAP controller mode signal
$\overline{\text{TRST}}$	Test reset	In	TAP controller reset
TAP_EN	TAP enable	In	n/a
TDO_EN	Test data output enable	Out	n/a
TLMSEL	TLM selected	Out	n/a

Test reset ($\overline{\text{TRST}}$) is an optional JTAG signal used in the e500 to reset the TAP controller asynchronously. This signal is not used during normal operation. It is recommended that $\overline{\text{TRST}}$ be asserted and negated coincident with the assertion of the $\overline{\text{HRESET}}$ signal to ensure that the test logic has no affect on the normal operation of the core complex.

$\overline{\text{TRST}}$ must be asserted sometime during power-up for JTAG logic initialization. Note that if $\overline{\text{TRST}}$ is tied low, unnecessary power is consumed.

8.2.1.1 JTAG Test Clock (TCK)—Input

The JTAG test clock (TCK) signal is the primary clock input for the test logic on the e500. This clock may be asynchronous with respect to all other core complex clocks. Following is the state meaning for the TCK input signal.

State Meaning Asserted/Negated—This input should be driven by a free-running clock signal. Input signals to the test access port are sampled on the rising edge of TCK. TAP output signal changes occur on the falling edge of TCK. The test logic allows TCK to be stopped.

8.2.1.2 JTAG Test Data Input (TDI)—Input

State Meaning Asserted/Negated—The value present on the rising edge of TCK is loaded into the selected JTAG test instruction or data register.

8.2.1.3 JTAG Test Data Output (TDO)—Output

State Meaning Asserted/Negated—The contents of the selected internal instruction or data register are shifted out onto this signal on the falling edge of TCK. The TDO signal remains in a quiet state except when scanning of data is in progress.

8.2.1.4 JTAG Test Mode Select (TMS)—Input

State Meaning Asserted/Negated—This signal is decoded by the internal JTAG TAP controller to determine the primary operation of the test support circuitry.

8.2.1.5 JTAG Test Reset ($\overline{\text{TRST}}$)—Input

State Meaning Asserted—This input causes asynchronous initialization of the internal JTAG test access port controller. Note that the signal must be asserted during the assertion of HRESET to properly initialize the JTAG test access port.

Negated—Indicates normal operation.

8.2.1.6 TAP Enable (TAP_EN)—Input

TAP_EN is used by the TAP linking module (TLM) logic external to the core complex to select the core complex TAP module. When there is no TLM connected to the TAP the TAP_EN is tied high.

State Meaning Asserted—a valid TMS signal is applied to the TAP controller.

Negated—logic low is applied to the TAP controller.

8.2.1.7 Test Data Out Enable (TDO_EN)—Output

TDO provides feedback to the external TAP linking module logic.

State Meaning Asserted—valid data is available on TDO

Negated—value of TDO is meaningless

8.2.1.8 TLM Selected (TLMSEL)—Output

TLMSEL provides feedback to the external TAP linking module logic.

State Meaning Asserted—the core complex is currently executing a TLM TAP instruction.

Negated—the core complex is not currently executing a TLM TAP instruction

8.3 Unconditional Debug Event

Book E defines a signal called unconditional debug event (UDE) which the core complex accepts as an input. By default, asserting UDE sets DBSR[UDE], and, if MSR[DE] is set, causes a debug interrupt to be taken. Several bits in the debug control register (DCR) can be used to override this behavior.

Setting DCR[ude_override] prevents DCR from signalling a UDE to the core complex.

Setting DCR[ude_hardstop] puts the core complex into the power-down state when UDE is set as shown in Table 8-4.

Table 8-4. Hardstop and Override Effects

ude_hardstop_en	ude_override	Core Complex Action
0	0	take UDE interrupt
0	1	go to halt
1	1	go to power down

DCR[runn_trig] = 0x2 causes the run counter to start counting.

DCR[interrupt_halt_en]. Corresponds to 19 IVORs. If a bit is set, device goes into halted state instead of exception handler.

8.4 Book E Debug Events

Debug events cause debug exceptions to be recorded in the DBSR (see Section 2.13.2, “Debug Status Register (DBSR)”). Except for an unconditional debug event (UDE), the specific event type must be enabled by corresponding bits in the debug control registers (DBCRC[0–2]) for any debug event to set a DBSR bit and thereby cause a debug exception. Setting a DBSR bit causes a debug interrupt only if debug interrupts are enabled by setting MSR[DE].

When MSR[DE] is cleared, some debug events are not recorded; that is, no DBSR bit is set by the event. However, some debug events can cause exceptions and set DBSR bits regardless of the state of MSR[DE]. The debug interrupts resulting from such exceptions are delayed until MSR[DE] is set (unless they have been cleared from DBSR in the meantime).

Any time a DBSR bit can be set while MSR[DE] is cleared, the imprecise debug event bit (DBSR[IDE]), is also set. DBSR[IDE] indicates that the associated debug exception bit in the DBSR was set while debug interrupts were disabled. Debug interrupt handler software can use this bit to interpret the address recorded in the critical save/restore register 0 (CSRR0). If DBSR[IDE] is cleared, then CSRR0 contains the address of the instruction causing the debug exception. If DBSR[IDE] is set, CSRR0 contains the address of the

instruction following the one that enabled the delayed debug interrupt (by setting MSR[DE]).

Debug interrupts are ordered with respect to other interrupt types (see Section 5.11.1, “e500 Interrupt Priorities”).

Debug exceptions are prioritized with respect to other exceptions (see Section 5.11, “Exception Priorities”).

The eight types of debug events are listed in Table 8-5 and discussed below.

Table 8-5. Debug Events

Event Type	Description	Section
Instruction address compare	Each instruction address is compared in a specific way with a specific value. A debug event occurs when they match.	8.4.1
Data address compare	Each data address is compared with a value. A debug event occurs when they match.	8.4.2
Trap	A debug event occurs when a trap is set.	8.4.3
Branch taken	A debug event occurs when any branch is taken.	8.4.4
Instruction complete	A debug event occurs when any instruction completes.	8.4.5
Interrupt taken	A debug event occurs when an interrupt is taken.	8.4.6
Return	A debug event occurs when a return from interrupt is taken.	8.4.7
Unconditional	A debug event occurs whenever this instruction is executed.	8.4.8

8.4.1 Instruction Address Compare Debug Event

One or more instruction address compare debug events (IAC1, IAC2, IAC3, or IAC4) (IAC3 and IAC4 are not supported on the e500) occur if they are enabled and execution is attempted of an instruction at an address that meets the criteria specified in DBCR0, DBCR1, and the IAC registers.

8.4.1.1 Instruction Address Compare User/Supervisor Mode

The debug control registers specify user/supervisor modes as follows:

- DBCR1[IAC1US] specifies whether IAC1 debug events can occur in user mode or supervisor mode, or both.
- DBCR1[IAC2US] specifies whether IAC2 debug events can occur in user mode or supervisor mode, or both.

8.4.1.2 EffectiveAddress Mode

The debug control registers specify effective address modes as follows:

- DBCR1[IAC1ER] specifies whether effective addresses alone, effective addresses and MSR[IS] cleared, or effective addresses and MSR[IS] set are used in determining an address match on IAC1 debug events.
- DBCR1[IAC2ER] specifies whether effective addresses alone, effective addresses and MSR[IS] cleared, or effective addresses and MSR[IS] set are used in determining an address match on IAC2 debug events.

8.4.1.3 Instruction Address Compare Mode

The debug control registers specify instruction address compare modes as follows:

- DBCR1[IAC12M] specifies whether all or some of the bits of the address of the instruction fetch must match the contents of IAC1 or IAC2, whether the address must be inside a specific range specified by IAC1 and IAC2 or outside a specific range specified by IAC1 or IAC2 to trigger a corresponding debug event.

The four instruction address compare modes are described in Table 8-6.

Table 8-6. Instruction Address Compare Modes

Mode Name	Description
Exact address compare	If the 32-bit address of the instruction fetch is equal to the value in the enabled IAC register, an instruction address match occurs.
Address bit match	For IAC1 and IAC2 debug events, if the address of the instruction fetch access, ANDed with the contents of IAC2, are equal to the contents of the IAC1, also ANDed with the contents of the IAC2, an instruction address match occurs.
Inclusive address range compare mode	For IAC1 and IAC2 debug events, if the 32-bit address of the instruction fetch is greater than or equal to the contents of the IAC1 and less than the contents of the IAC2, an instruction address match occurs.
Exclusive address range compare mode	For IAC1 and IAC2 debug events, if the 32-bit address of the instruction fetch is less than the contents of the IAC1 or greater than or equal to the contents of the IAC2, an instruction address match occurs.

Section 2.13.1, “Debug Control Registers,” describes DBCR0 and DBCR1 and the modes of for detecting IAC register debug events. Instruction address compare debug events can occur regardless of the value of MSR[DE].

When an instruction address compare debug event occurs, the corresponding DBSR[IACn] bits are set to record the debug exception. If MSR[DE] is cleared, DBSR[IDE] is also set to capture the imprecise debug event.

If MSR[DE] is set (debug interrupts are enabled) at the time of the instruction address compare debug exception, a debug interrupt occurs immediately (if no higher priority exception has caused an interrupt). The execution of the instruction causing the exception is suppressed, and CSRR0 is set to the address of the excepting instruction.

If MSR[DE] is cleared (debug interrupts are disabled) at the time of the instruction address compare debug exception, a debug interrupt does not occur, and the instruction completes execution (provided the instruction is not causing some other exception that generates an enabled interrupt).

Later, if the debug exception has not been reset by clearing the appropriate DBSR[IAC n], bits and MSR[DE] is set, a delayed debug interrupt occurs. In this case, CSRR0 contains the address of the instruction following the one that enabled the debug interrupt (by setting MSR[DE]). Software in the debug interrupt handler can observe DBSR[IDE] to determine how to interpret the CSRR0 value.

8.4.2 Data Address Compare Debug Event

One or more data address compare debug events (DAC1R, DAC1W, DAC2R, DAC2W) occur if they are enabled, execution is attempted of a data access instruction, and the type, address, and possibly even the data value of the data access meet the criteria specified in DBCR0, DBCR2, DAC1, and DAC2.

8.4.2.1 Data Address Compare Read/Write Enable

DBCR0[DAC1] specifies whether DAC1R debug events can occur on read-type data accesses and whether DAC1W debug events can occur on write-type data accesses.

DBCR0[DAC2] specifies whether DAC2R debug events can occur on read-type data accesses and whether DAC2W debug events can occur on write-type data accesses.

All load instructions are considered reads with respect to debug events, and all store instructions are considered writes with respect to debug events. In addition, cache management instructions, and certain special cases, are handled as follows.

- **dcbt[e]**, **dcbtst[e]**, **icbt[e]**, and **icbi[e]** are all considered reads with respect to debug events. Note that **dcbt[e]**, **dcbtst[e]**, and **icbt[e]** are treated as no-operations when they report data storage or data TLB miss exceptions, instead of being allowed to cause interrupts. However, these instructions are allowed to cause debug interrupts, even when no-op would have been asserted due to a data storage or data TLB miss exception.
- **dcbz[e]**, **dcbi[e]**, **dcbf[e]**, and **dcbst[e]** are all considered writes with respect to debug events. Note that **dcbf[e]** and **dcbst[e]** are considered reads with respect to data storage exceptions because they do not change the data at a given address. However, because execution of these instructions may generate write activity on the processor's data bus, they are treated as writes with respect to debug events.
- Indexed-string instructions (**lswx** and **stswx**) for which the XER field specifies zero bytes as the length of the string are treated as no-ops and are not allowed to cause data address compare debug events.

8.4.2.2 Data Address Compare User/Supervisor Mode

User/supervisor mode options in data address compare debug events occur as follows:

- DBCR2[DAC1US] specifies whether DAC1R and DAC1W debug events can occur in user mode, supervisor mode, or both.
- DBCR2[DAC2US] specifies whether DAC2R and DAC2W debug events can occur in user mode, supervisor mode, or both.

8.4.2.3 Effective Address Mode

Effective address mode options in debug events occur as follows:

- DBCR2[DAC1ER] specifies whether effective addresses alone, effective addresses and MSR[DS] cleared, or effective addresses and MSR[DS] set, are used to determine an address match on DAC1R and DAC1W debug events.
- DBCR2[DAC2ER] specifies whether effective addresses alone, effective addresses and MSR[DS] cleared, or effective addresses and MSR[DS] set, are used to determine an address match on DAC2R and DAC2W debug events.

8.4.2.4 Data Address Compare (DAC) Mode

DBCR2[DAC12M] specifies whether all or some of the bits of the address of the data access must match the DAC1 or DAC2 contents, whether the address must be inside a range specified by DAC1 and DAC2 or outside a range specified by DAC1 and DAC2 for a DAC1R, DAC1W, DAC2R, or DAC2W debug event to occur.

The four data address compare modes are discussed in Table 8-7.

Table 8-7. Data Address Compare Modes

Mode Name	Description
Exact address compare	If the 32-bit address of the data access is equal to the value in the enabled DAC _n , a data address match occurs.
Address bit match	If the address of the data access, ANDed with the contents of the DAC2, are equal to the contents of the DAC1, also ANDed with the contents of the DAC2, a data address match occurs.
Inclusive address range compare	If the 32-bit address of the data access is greater than or equal to the contents of DAC1 and less than the contents of DAC2, a data address match occurs.
Exclusive address range compare	If the 32-bit address of the data access is less than the contents of DAC1 or greater than or equal to the contents of DAC2, a data address match occurs.

Section 2.13.1, “Debug Control Registers,” describes DBCR0 and DBCR2 and the modes for detecting DAC debug events. DAC debug events can occur regardless of the value of MSR[DE].

When a DAC debug event occurs, the corresponding DBSR[DAC1R], DBSR[DAC1W], DBSR[DAC2R], or DBSR[DAC2W] is set to record the exception. If MSR[DE] is cleared, then DBSR[IDE] is also set to capture the imprecise debug event.

If MSR[DE] is set (debug interrupts are enabled) a DAC debug exception causes the following events.

- A debug interrupt is taken immediately (if no higher priority exception has caused an interrupt)
- Execution of the instruction causing the exception is suppressed
- CSSR0 is loaded with the address of the excepting instruction

Depending on the type of instruction and the alignment of the access, the instruction causing the exception may have been partially executed (see Section 5.9, “Partially Executed Instructions”).

If MSR[DE] is cleared (debug interrupts are disabled) when the DAC debug exception occurs, a debug interrupt is not taken and the instruction completes normally (provided the instruction is not causing some other exception that generates an enabled interrupt). Also, DBSR[IDE] is set to indicate that the exception occurred while debug interrupts were disabled.

Later, if MSR[DE] is set, and the debug exception has not been reset by clearing DBSR[DAC1R], DBSR[DAC1W], DBSR[DAC2R], DBSR[DAC2W], a delayed debug interrupt occurs. In this case, CSSR0 contains the address of the instruction following the one that enabled the debug interrupt (by setting MSR[DE]). The debug interrupt handler can observe DBSR[IDE] to determine how to interpret the CSSR0 value.

8.4.3 Trap Debug Event

A trap debug event (TRAP) occurs if DBCR0[TRAP] is set (trap debug events are enabled) and a trap instruction (**tw**, **twi**, **td**, or **tdi**) is executed and the trap conditions specified by the instruction are met. The event can occur regardless of the value of MSR[DE].

When a trap debug event occurs, DBSR[TRAP] is set to capture the debug exception. If MSR[DE] is cleared, DBSR[IDE] is also set to record the imprecise debug event.

If MSR[DE] is set at the time of the trap debug exception, a debug interrupt occurs immediately (if no higher priority exception has caused an interrupt), and CSSR0 is set to the address of the excepting instruction.

If MSR[DE] is cleared (debug interrupts are disabled) at the time of the trap debug exception, a debug interrupt is not taken, and a trap exception type program interrupt occurs instead.

Later, if MSR[DE] is set, and the debug exception has not been reset by clearing DBSR[TRAP], a delayed debug interrupt occurs. In this case, CSSR0 contains the address

of the instruction following the one that enabled the debug interrupt (by setting MSR[DE]). The debug interrupt handler can observe DBSR[IDE] to determine how to interpret the CSSR0 value.

8.4.4 Branch Taken (BRT) Debug Event

A branch taken (BRT) debug event occurs if both MSR[DE] and DBCR0[BRT] are set (branch taken debug events are enabled), and execution is attempted of a branch instruction whose direction is taken (an unconditional branch or a conditional branch whose branch condition is met).

Because branch instructions occur very frequently, branch taken debug events are not recognized if MSR[DE] is cleared when the branch instruction executes and thus DBSR[IDE] cannot be set by a BRT debug event. Allowing these common events to be recorded as exceptions in the DBSR while debug interrupts are disabled would cause an inordinate number of imprecise debug interrupts.

The following actions are taken when a branch taken debug event occurs.

- DBSR[BRT] is set (to capture the debug exception)
- a debug interrupt occurs immediately (if no higher priority exception has caused an interrupt)
- execution of the exception causing instruction is suppressed
- CSSR0 is set to the address of the excepting instruction

8.4.5 Instruction Complete Debug Event

An instruction complete debug event (ICMP) occurs when any instruction completes execution so long as MSR[DE] and DBCR0[ICMP] are both set (ICMP debug events are enabled). Note that no ICMP debug event occurs if execution of an instruction is suppressed because it caused some other interrupt-generating exception. The **sc** instruction does not fall into the category of an instruction whose execution is suppressed, because the instruction actually completes execution and then generates a system call interrupt. In this case, the instruction complete debug exception is also set.

Instruction complete debug events are not recognized if MSR[DE] is cleared at the time of the instruction execution. DBSR[IDE] cannot be set by an ICMP debug event because allowing the common instruction completion event to log an exception in the DBSR while debug interrupts are disabled would cause the debug interrupt handler software to receive an inordinate number of imprecise debug interrupts whenever debug interrupts were reenabled.

The following actions are taken when an instruction complete debug event occurs:

- DBSR[ICMP] is set (to record the debug exception)

- a debug interrupt occurs immediately (if no higher priority exception has caused an interrupt)
- CSSR0 is set to the address of the instruction following the one that caused the ICMP debug exception

8.4.6 Interrupt Taken (IRPT) Debug Event

An interrupt taken (IRPT) debug event occurs if DBCR0[IRPT] is set (IRPT debug events are enabled) and a non-critical interrupt occurs. IRPT debug events can occur regardless of the value of MSR[DE].

Only non-critical interrupts can cause an IRPT debug event because all critical interrupts automatically clear MSR[DE], and thus would always prevent the associated debug interrupt from occurring precisely. Also, debug interrupts themselves are critical interrupts, so any additional debug interrupt (for a second debug event) would always set the additional DBSR[IRPT] exception when it entered the debug interrupt handler. At this point, the debug interrupt handler could not determine if the second IRPT debug event was related to the original event.

When an IRPT debug event occurs, DBSR[IRPT] is set to capture the debug exception. If MSR[DE] is cleared, DBSR[IDE] is also set to record the imprecise debug event.

If MSR[DE] is set (enabling debug interrupts) at the time of the interrupt taken debug event, the following occurs:

- a debug interrupt occurs immediately, if no higher priority exception has caused an interrupt
- CSSR0 is set to the address of the non-critical interrupt vector that caused the IRPT debug event. No instructions at the non-critical interrupt handler will have been executed.

If debug interrupts are disabled (MSR[DE] cleared) when the IRPT debug event occurs, no interrupt is generated. However, if the debug exception has not been reset by clearing DBSR[IRPT], a delayed debug interrupt occurs when interrupts are reenabled (MSR[DE] is set). In this case, CSSR0 contains the address of the instruction following the one that set MSR[DE]. The interrupt handler can observe DBSR[IDE] to determine how to interpret CSSR0.

8.4.7 Return Debug Event

A return debug event (RET) occurs if DBCR0[RET] is set (enabling return debug events) and an attempt is made to execute an **rfi**.

Results from executing an **rftci** while DBCR0[RET] is set are implementation-dependent. If DBCR0[RET] is set, the e500 does the following when an **rftci** is executed:

- If MSR[DE] is set, a debug interrupt is generated.
- If MSR[DE] is cleared, no debug interrupt is generated and no debug event is logged.

When a return debug event occurs, DBSR[RET] is set to capture the debug exception. If MSR[DE] is cleared when the **rfi** executes (before the MSR is updated by the **rfi**), DBSR[IDE] is also set to record the imprecise debug event.

If MSR[DE] is set at the time of the return debug exception, the following events occur.

- a debug interrupt is taken immediately (unless the **rfi** or **rfci** causing the return debug event itself clears MSR[DE], or a higher priority exception has caused an interrupt)
- CSSR0 is loaded with the address of the instruction that would have executed next had the interrupt not occurred

If MSR[DE] is cleared (either at the time of the execution of the **rfi**, or after the MSR is updated by the **rfi**) at the time of the return debug exception, a debug interrupt does not occur.

Provided the debug exception has not been reset by clearing DBSR[RET], a delayed imprecise debug interrupt occurs when MSR[DE] is set. In this case, CSSR0 contains the address of the instruction following the one that set MSR[DE]. The interrupt handler can observe DBSR[IDE] to determine how to interpret the value in CSSR0 unless MSR[DE] was cleared by the **rfi**. In that case DBSR[IDE] will not have been set and the software cannot determine that the interrupt was precise.

8.4.8 Unconditional Debug Event

An unconditional debug event (UDE) occurs when the unconditional debug event (UDE) signal is activated by the debug mechanism. The UDE is the only debug event that does not have a corresponding enable bit for the DBCR0 event (hence the name of the event). The UDE can occur regardless of the value of MSR[DE].

When an unconditional debug event occurs, DBSR[UDE] is set to capture the debug exception. If MSR[DE] is cleared, DBSR[IDE] is also set to capture the imprecise debug event.

If MSR[DE] is set (enabling debug interrupts) at the time of the unconditional debug exception, the following occurs.

- a debug interrupt is taken immediately, if no higher priority exception has caused an interrupt
- CSSR0 is set to the address of the instruction that would have executed next had the interrupt not occurred

If the UDE occurs when debug interrupts are disabled, no debug interrupt occurs.

Provided the debug exception has not been reset by clearing DBSR[UDE], a delayed debug interrupt occurs when MSR[DE] is set. In this case, CSRR0 contains the address of the instruction following the one that set MSR[DE]. The debug interrupt handler can observe DBSR[IDE] to determine how to interpret the CSSR0 value.

Part II

e500 Core Complex

This part describes the features of the e500 core complex that comprise its memory subsystem and auxiliary features. It contains the following chapters:

- Chapter 9, “Timer Facilities,” describes the Book E-defined timer facilities implemented in the e500 core. These resources include the time base (TB), decrements (DEC), fixed-interval timer (FIT), and watchdog timer.
- Chapter 10, “Auxiliary Processing Units (APUs),” describes APUs implemented on the e500, such as the **isel** instruction, performance monitor, signal processing engine, branch target buffer (BTB) locking, cache block lock and unlock, and machine check APUs.
- Chapter 11, “Cache and MMU Background,” describes aspects of the memory hierarchy and the memory management model defined by Book E. It also describes the Motorola implementation standards, providing context for the wide flexibility of Book E architecture memory and cache definitions.
- Chapter 12, “L1 Caches,” describes the organization of the on-chip level-one instruction and data caches, cache coherency protocols, cache control instructions, and various cache operations. It describes the interaction that occurs in the memory subsystem, which consists of the memory management unit (MMU), the caches, the load/store unit (LSU), and the core complex bus (CCB). The chapter also describes the replacement algorithms used for each of the L1 caches.
- Chapter 13, “Memory Management Units,” describes the implementation details of the e500 core complex MMU relative to the Book E architecture and the Motorola Book E standards.

Chapter 9

Timer Facilities

This chapter describes the Book E-defined timer facilities implemented in the e500 core. These resources include the time base (TB), decrementer (DEC), fixed-interval timer (FIT), and watchdog timer.

9.1 Timer Facilities

The TB, DEC, FIT, and watchdog timer provide timing functions for the system. All of these must be initialized during start-up.

- The time base provides a long-period counter driven by a frequency that is implementation-dependent.
- The decrementer, a counter that is updated at the same rate as the TB, provides a means of signaling an exception after a specified amount of time has elapsed unless:
 - DEC is altered by software in the interim, or
 - The TB update frequency changes

The DEC is typically used as a general-purpose software timer.

- The fixed-interval timer is essentially a selected bit of the TB, which provides a means of signalling an exception whenever the selected bit transitions from 0 to 1, in a repetitive fashion. The fixed-interval timer is typically used to trigger periodic system maintenance functions. Software may select one of four bits in the TB to serve as the fixed-interval timer. Which bits may be selected depends on the implementation.
- The watchdog timer is also a selected bit of the TB, which provides a means of signalling a critical class exception whenever the selected bit transitions from 0 to 1. In addition, if software does not respond in time to the initial exception (by clearing the associated status bits in the TSR before the next expiration of the watchdog timer interval), then a watchdog timer-generated processor reset may result, if so enabled. The watchdog timer is typically used to provide a system error recovery function.

The relationship of these timer facilities to each other is shown in Figure 9-1.

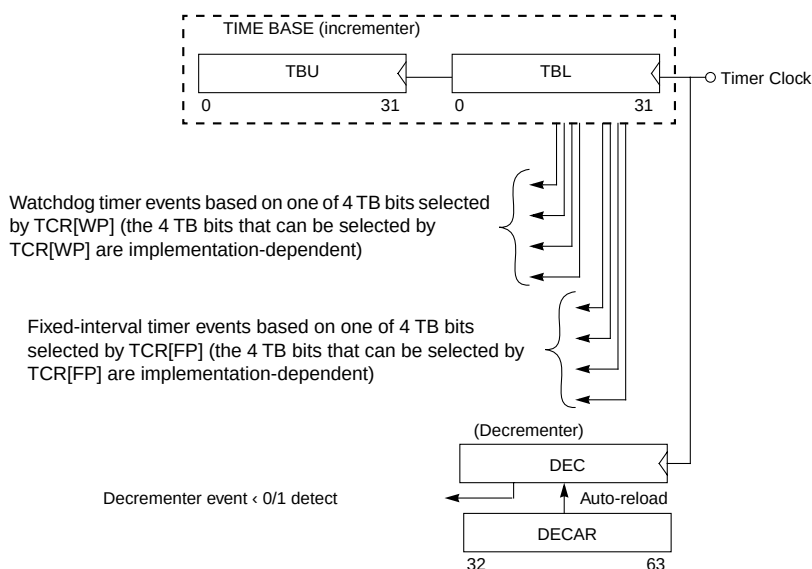


Figure 9-1. Relationship of Timer Facilities to TB

9.2 Timer Registers

This section describes registers that are used by the timer facilities.

9.2.1 Hardware Implementation-Dependent Register 0

The e500 implements the HID0[TBEN] which is used to enable the time base and HID0[SEL_TBCLK] which selects the time base clock. For more information, see Section 2.10.1, “Hardware Implementation-Dependent Register (HID0).”

Section 9.3, “The e500 Timer Implementation,” describes how this bit interacts with other registers.

9.2.2 Timer Control Register (TCR)

The TCR, shown in Figure 9-2, provides control information for the on-chip timer of the core complex. The core complex implements two fields not specified in Book E: TCR[WPEXT] and TCR[FPEXT].

The timer control register (TCR) controls decrementer (see Section 9.3.1, “Decrementer Register”), fixed-interval timer (see Section 9.4, “Fixed-Interval Timer”), and watchdog timer (see Section 9.5, “Watchdog Timer”) options.

TCR contents can be read into a GPR using **mf spr**. The GPR contents can be written to the TCR using **mt spr**.

	32	33	34	35	36	37	38	39	40	41	42	43		46	47		50	51		63
Field	WP	WRC	WIE	DIE	FP	FIE	ARE	—	WPEXT	FPEXT	—									
Reset	All zeros																			
R/W	R/W																			
SPR	SPR 340																			

Figure 9-2. Timer Control Register (TCR)

Table 9-1 describes the TCR fields.

Table 9-1. TCR Field Descriptions

Bits	Name	Description
32–33	WP	Watchdog timer period. When concatenated with WPEXT, specifies one of 64 bit locations of the time base used to signal a watchdog timer exception on a transition from 0 to 1. WPEXT[0–3],WP[0–1] == 0b000000 selects TBU[0] WPEXT[0–3],WP[0–1] == 0b111111 selects TBL[63]
34–35	WRC	Watchdog timer reset control. Allows watchdog timer-caused resets. 0 When cleared, this bit has no effect on watchdog timer operation 1 Signals external logic to reset the core. When the watchdog timer times out the second time, TCR[WRC] is copied to TSR[WRS], and these signals are connected to external pins (). External logic must define these pins and act appropriately when set.
36	WIE	Watchdog timer interrupt enable 0 Watchdog timer interrupts disabled 1 Watchdog timer interrupts enabled
37	DIE	Decrementer interrupt enable 0 Decrementer interrupts disabled 1 Decrementer interrupts enabled
38–39	FP	Fixed interval timer period. When concatenated with FPEXT, specifies one of 64 bit locations of the time base used to signal a fixed-interval timer exception on a transition from 0 to 1. WPEXT[0–3],WP[0–1] == 0b000000 selects TBU[0] WPEXT[0–3],WP[0–1] == 0b111111 selects TBL[63]
40	FIE	Fixed interval interrupt enable 0 Fixed interval interrupts disabled 1 Fixed interval interrupts enabled
41	ARE	Auto-reload enable 0 Auto-reload disabled 1 Auto-reload enabled
42	—	TBD
43–46	WPEXT	Watchdog timer period extension (see above description for WP)
47–50	FPEXT	Fixed-interval timer period extension (see above description for FP)
50–63	—	Reserved, should be cleared.

9.2.3 Timer Status Register (TSR)

The 32-bit TSR contains status on timer events and the most recent watchdog timer-initiated processor reset.

	32	33	34	35	36	37	38	63
Field	ENW	WIS	WRS	DIS	FIS	—		
Reset	All zeros							
R/W	Set by hardware; read by using mfsprr , and cleared by using mtsprr by writing ones to any TSR bit positions to be cleared and zeros in all other bit positions. The write-data to the TSR is not direct data, but a mask. A 1 causes the bit to be cleared, and a 0 has no effect.							
SPR	SPR 336							

Figure 9-3. Timer Status Register (TSR)

Table 9-2 describes TSR fields.

Table 9-2. TSR Field Descriptions

Bits	Name	Description
32	ENW	Enable next watchdog time. (See Section 9.5, "Watchdog Timer.") 0 Action on next watchdog timer time-out is to set TSR[ENW]. 1 Action on next watchdog timer time-out is governed by TSR[WIS].
33	WIS	Watchdog timer interrupt status. (See Section 9.5, "Watchdog Timer.") 0 A watchdog timer event has not occurred. 1 A watchdog timer event has occurred. When MSR[CE] is set and TCR[WIE] is set, a watchdog timer interrupt is taken.
34–35	WRS	Watchdog timer reset status. (See Section 9.5, "Watchdog Timer") These two bits are set to one of three values when a reset is caused by the watchdog timer. These bits are undefined at power-up. 00 No watchdog timer reset has occurred. 01 Implementation-dependent reset information. 10 Implementation-dependent reset information. 11 Implementation-dependent reset information.
36	DIS	Decrementer interrupt status. (See Section 9.3.3, "Decrementer Operation.") 0 A decrementer event has not occurred. 1 A decrementer event occurred. When MSR[EE] is set and TCR[DIE] is set, a decrementer interrupt is taken.
37	FIS	Fixed-interval timer interrupt status. (See Section 9.4, "Fixed-Interval Timer.") 0 A fixed-interval timer event has not occurred. 1 A fixed-interval timer event has occurred. When MSR[EE] is set and TCR[FIE] is set, a fixed-interval timer interrupt is taken.
38–63	—	Reserved, should be cleared.

9.2.4 Time Base

The time base (TB) is composed of two 32-bit registers, the time base upper (TBU) concatenated on the right with the time base lower (TBL).

	32	63 32	63
Field	TBU		TBL
Reset	All zeros		All zeros
R/W	User read/Supervisor write		User read/Supervisor write
SPR	269 Read/ 285 Write		268 Read/ 284 Write

Figure 9-4. Timer Base Upper/Lower Registers (TBU/TBL)

The TB is interpreted as a 64-bit unsigned integer that is incremented periodically. Each increment adds 1 to the least-significant bit. The frequency at which the integer is updated is implementation-dependent.

The TB provides timing functions for the system. The TB is a volatile resource and must be initialized during start-up.

The TB provides a long-period counter driven by an implementation-dependent frequency.

TBU and TBL are accessed as follows:

- TBU contents can be read using **mfspir**. The contents of a GPR can be written to the TBU using **mtspir**.
- TBL contents can be read into a GPR using **mfspir**. The contents of GPR(RS) can be written to the TBL using **mtspir**.

There is no automatic initialization of the TB; system software must perform this initialization.

TBL increments until its value becomes 0xFFFF_FFFF ($2^{32}-1$). At the next increment, its value becomes 0x0000_0000 and TBU is incremented. This process continues until the value in the TBU becomes 0xFFFF_FFFF and the value in the TBL becomes 0xFFFF_FFFF, or the value in the TB is interpreted as 0xFFFF_FFFF_FFFF_FFFF ($2^{64}-1$). At the next increment, the value in TBU becomes 0x0000_0000 and the value in TBL becomes 0x0000_0000. There is no interrupt or other indication when this occurs.

The period of the TB depends on the driving frequency. As an order of magnitude example, suppose that the TB is driven by a frequency of 100 MHz divided by 32. Then the period of the TB would be as follows:

$$T_{TB} = 2^{64} \times 32 / 100 \text{ MHz} = 5.90 \times 10^{12} \text{ seconds}$$

This value is approximately 187,000 years.

The TB must be implemented such that the following requirements are satisfied.

- Loading a GPR from the TB shall have no effect on the accuracy of the TB.
- Storing a GPR to the TB replaces the value in the TB with the value in the GPR.

Book E does not specify a relationship between the frequency at which the TB is updated and other frequencies, such as the CPU clock or bus clock in a Book E system. The TB update frequency is not required to be constant. One of the following is required to ensure that system software can keep time of day and operate interval timers:

- The system provides an (implementation-dependent) interrupt to software whenever the update frequency of the TB changes and a way to determine the current update frequency.
- The update frequency of the TB is under the control of the system software.

Implementations must provide a means for either preventing the TB from incrementing or preventing the TB from being read in problem state (MSR[PR] is set). If the means is under software control, the TB must be accessible only in privileged state (MSR[PR] is clear).

NOTE

Disabling the TB or making reading the time base privileged prevents the TB from being used to implement a covert channel in a secure system.

The requirements stated above for the TB apply also to any other SPRs that measure time and can be read in problem state (for example, performance monitor registers).

NOTE

If the operating system initializes the TB on power-on to some reasonable value and the update frequency of the TB is constant, the TB can be used as a source of values that increase at a constant rate, such as for time stamps in trace entries.

Even if the update frequency is not constant, values read from the TB are monotonically increasing (except when the TB wraps from $2^{64}-1$ to 0). If a trace entry is recorded each time the update frequency changes, the sequence of TB values can be post-processed to become actual time values.

Successive readings of the TB may return identical values.

See Section 9.2.4.3, “Computing Time of Day from the Time Base,” for ways to compute time of day in POSIX format from the TB.

It is intended that the TB be useful for timing reasonably short sequences of code (a few hundred instructions) and for low-overhead time stamps for tracing.

9.2.4.1 Writing the TB

It is not possible to write the entire 64-bit TB using a single instruction. The TB can be written by a sequence such as:

```
lwz      Rx,upper # load 64-bit value for
lwz      Ry,lower # TB into Rx and Ry
addi     Rz,R0,0
mtspr    TBL,Rz   # force TBL to 0
mtspr    TBU,Rx   # set TBU
mtspr    TBL,Ry   # set TBL
```

Provided that no interrupts occur while the last three instructions are being executed, loading 0 into time base lower prevents the possibility of a carry from time base lower to TBU while the TB is being initialized.

9.2.4.2 Reading the TB

The 64-bit time base cannot be read in a single instruction. **mfspir rD,TBL** reads the lower half into a GPR, and **mfspir rD,TBU** reads the upper half. Because a carry from TBL to TBU can occur between reads, a sequence such as the following is necessary.

```
loop:
    mfspir Rx,TBU #load from TBU
    mfspir Ry,TBL #load from TBL
    mfspir Rz,TBU #load from TBU
    cmp    cr0,0,Rz,Rx #see if 'old' = 'new'
    bc     4,2,loop #loop if carry occurred
```

The comparison and loop ensure that a consistent pair of values is obtained.

9.2.4.3 Computing Time of Day from the Time Base

Because the TB update frequency is implementation-dependent, the algorithm for converting the current value in the TB to time of day is also implementation-dependent.

For example, assume the TB is constantly incremented once for every 32 cycles of a 100 MHz CPU instruction clock. What is wanted is the pair of 32-bit values comprising a POSIX standard clock: the number of whole seconds that have passed since midnight January 1, 1970, and the remaining fraction of a second expressed as a number of nanoseconds.

Assume the following:

- The value 0 in the TB represents the start time of the POSIX clock (if this is not true, a simple 64-bit subtraction will make it so).
- The integer constant `ticks_per_sec` contains the following value, which is the number of times the TB is updated each second:

$$100\text{MHz} / 32 = 3,125,000$$

Timer Registers

- The integer constant `ns_adj` contains the following value, which is the number of nanoseconds per tick of the TB:

$$1,000,000,000 / 3,125,000 = 320$$

The POSIX clock can be computed with an instruction sequence such as the following:

```
# Read Time Base
loop:
    mfspr Rx,TBU #load from TBU into Rx
    mfspr Ry,TBL #load from TBL into Ry
    mfspr Rz,TBU #load from TBU into Rz
    cmp CR0,0,Rz,Rx #see if 'old TBU' = 'new TBU'
    bc 4,2,loop #loop if carry occurred
    rldimi Ry,Rx,32,0#splice TBU & TBL into Ry

#
# Compute POSIX clock
#
    lwz Rx,ticks_per_sec
    divd Rz,Ry,Rx #Rz = whole seconds
    stw Rz,posix_sec
    mulld Rz,Rz,Rx #Rz = quotient * divisor
    sub Rz,Ry,Rz #Rz = excess ticks
    lwz Rx,ns_adj
    mulld Rz,Rz,Rx #Rz = excess nanoseconds
    stw Rz,posix_ns
```

In the absence of a **divd** instruction (see Appendix C, “Guidelines for 32-Bit Book E”), direct implementation of the algorithm given above is awkward.

Such division can be avoided entirely if a time of day clock in POSIX format is updated at least once each second.

Assume the following:

- The operating system maintains the following variables:
 - *posix_tb* (64 bits)
 - *posix_sec* (32 bits)
 - *posix_ns* (32 bits)

These variables hold the value of the TB and the computed POSIX second and nanosecond values from the last time the POSIX clock was computed.

- The operating system arranges for an interrupt to occur at least once per second, at which time it recomputes the POSIX clock values.
- The integer constant *billion* contains the value 1,000,000,000.

The POSIX clock can be computed with an instruction sequence such as this:

```
loop:
    mfspr Rx,TBU #Rx = TBU
    mfspr Ry,TBL #Ry = TBL
    mfspr Rz,TBU #Rz = 'new' TBU value
```

```

        cmp      CR0,0,Rz,Rx  #see if 'old' = 'new'
        bc      4,2,loop      #loop if carry occurred
# now have 64-bit TB in Rx and Ry
        lwz      Rz,posix_tb+4
        sub      Rz,Ry,Rz      #Rz = delta in ticks
        lwz      Rw,ns_adj
        mullw     Rz,Rz,Rw      #Rz = delta in ns
        lwz      Rw,posix_ns
        add      Rz,Rz,Rw      #Rz = new ns value
        lwz      Rw,billion
        cmp      CR0,0,Rz,Rw  #see if past 1 second
        bc      12,0,nochange  #branch if not
        sub      Rz,Rz,Rw      #adjust nanoseconds
        lwz      Rw,posix_sec
        addi     Rw,Rw,1        #adjust seconds
        stw      Rw,posix_sec  #store new seconds
nochange:
        stw      Rz,posix_ns    #store new ns
        stw      Rx,posix_tb    #store new time base
        stw      Ry,posix_tb+4

```

Note that the upper half of the TB does not participate in the calculation to determine the new POSIX time of day. This is correct as long as the time change does not exceed one second.

9.2.4.3.1 Non-Constant Update Frequency

In a system in which the update frequency of the TB may change over time, it is not possible to convert an isolated TB value into time of day. Instead, a TB value has meaning only with respect to the current update frequency and the time of day that the update frequency was last changed. Each time the update frequency changes, either the system software is notified of the change by way of an interrupt (see Chapter 5, “Interrupts and Exceptions,” describes register behavior caused by the decremter interrupt), or the change was instigated by the system software itself. At each such change, the system software must compute the current time of day using the old update frequency, compute a new value of *ticks_per_sec* for the new frequency, and save the time of day, TB value, and tick rate. Subsequent calls to compute time of day use the current TB value and the saved data.

9.3 The e500 Timer Implementation

The e500 time base works as follows:

- If HID0[TBEN] = 0, the time base is static (no counting)
- If HID0[TBEN] = 1 and HID0[SEL_TBCLK] = 0, the time base is updated every 8 bus clocks
- If HID0[TBEN] = 1 and HID0[SEL_TBCLK] = 1, the time base is updated on the rising edge of core_TBCLK (sampled at bus rate). The maximum supported

frequency can be found in the electrical specifications, but this value is approximately 25% of the bus clock frequency.

The decrementer, TBL, and TBU are updated in that order during three successive internal processor clock cycles.

The core output signals `core_wrs` reflect the value of `TSR[WRS]`. The intention is to signal to the system that a watchdog reset event has occurred. The system can then implement a reset strategy. The core can be reset by asserting `core_hreset`. No automatic resetting is done when a watchdog reset occurs.

9.3.1 Decrementer Register

The contents of the decrementer (DEC) can be read into bits 32–63 of a GPR using `mfspr`, clearing bits 0–31. The contents of a GPR can be written to the decrementer using `mtspr`.

	32	63
Field	Decrementer value	
Reset	All zeros	
R/W	R/W	
SPR	SPR 22	

Figure 9-5. Decrementer Register (DEC)

9.3.2 Decrementer Auto-Reload Register (DECAR)

The decrementer auto-reload register (DECAR) supports the auto-reload feature of the decrementer. The DECAR contents cannot be read. The contents of a GPR can be written to DECAR using `mtspr`.

	32	63
Field	Decrementer auto-reload value	
Reset	All zeros	
R/W	R/W	
SPR	SPR 54	

Figure 9-6. Decrementer Auto-Reload Register (DECAR)

9.3.3 Decrementer Operation

The DEC decrements at the same rate that the TB increments. A decrementer event occurs when a decrement occurs on a decrementer value of 0x0000_0001. At this point, the decrementer has the following modes of operation, depending on whether the auto-reload function is enabled.

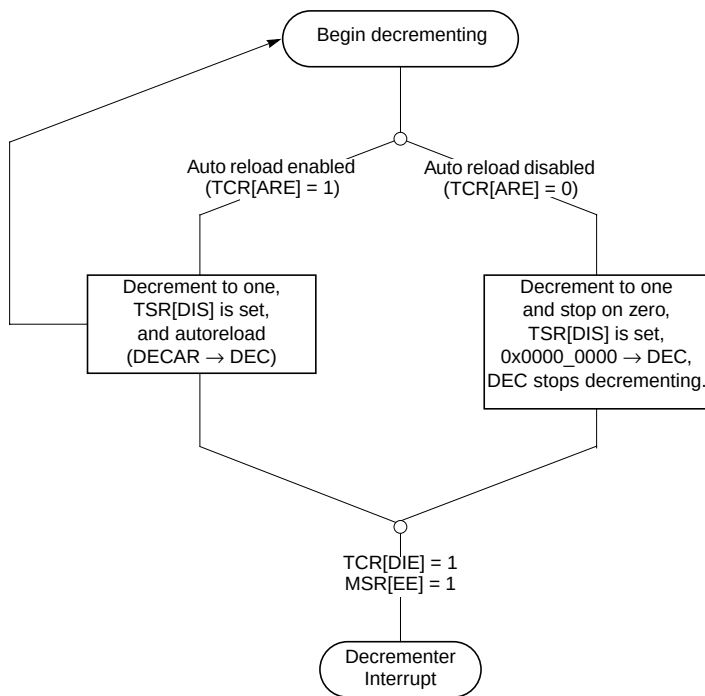


Figure 9-7. Decrementer Operation

- TCR[ARE] is cleared (auto-reload function disabled): Decrement to one and stop on zero.
- When the decrement event occurs, the decrementer interrupt status bit (TSR[DIS]) is set and the value 0x0000_0000 is placed into the DEC, which then stops decrementing. It remains at zero until software reloads it using an **mtspr** instruction. If the decrementer interrupt enabled (TCR[DIE] = 1 and MSR[EE] = 1), the interrupt is taken. Section 5.7.9, “Decrementer Interrupt,” describes register behavior caused by the decrementer interrupt.
- TCR[ARE] is set (auto-reload function enabled): Decrement to one and auto-reload. When the decrementer event occurs, TSR[DIS] is set and the DECAR contents are placed into DEC, which continues decrementing from the reloaded value. If the decrementer interrupt enabled (TCR[DIE] = 1 and MSR[EE] = 1), the interrupt is taken. Section 5.7.9, “Decrementer Interrupt,” describes register behavior caused by the decrementer interrupt.

The decrements interrupt handler must reset TSR[DIS] to avoid taking another, redundant decrements interrupt once MSR[EE] is re-enabled (assuming TCR[DIE] is not cleared instead). This is done by writing a word to TSR using **mtspr** with a 1 in the bit corresponding to TSR[DIS] (and any other bits that are to be cleared) and 0 in all other bits. The write-data to the TSR is not direct data, but a mask. A 1 causes the bit to be cleared, and a 0 has no effect.

Forcing the decrements to 0 using **mtspr** does not cause a decrements exception; however, decrementing which was in progress at the instant of the **mtspr** may cause the exception. To eliminate the decrements as a source of exceptions, set TCR[DIE] to 0 (clear the decrements interrupt enable bit).

To eliminate all decrements activity, the procedure is as follows:

1. Write 0 to TCR[DIE]. This prevents decrements activity from causing exceptions.
2. Write 0 to TCR[ARE] to disable the decrements auto-reload.
3. Write 0 to DEC. This halts decrementing. Although this action does not cause a decrements exception to be set in TSR[DIS], a near simultaneous decrement may have done so.
4. Write 1 to TSR[DIS]. This action clears TSR[DIS] (see Section 9.2.3, “Timer Status Register (TSR)”) and removes any pending decrements exception. Because the decrements is frozen at zero, no further decrements events are possible.

If the auto-reload feature is disabled (TCR[ARE] is cleared), when the DEC reaches zero, it stays there until software reloads it using the **mtspr** instruction.

On reset, TCR[ARE] is cleared, which disables the auto-reload feature.

9.4 Fixed-Interval Timer

The fixed-interval timer (FIT) is a mechanism for providing timer interrupts with a repeatable period, to facilitate system maintenance. It is similar in function to an auto-reload decrements, except that there are fewer selections of interrupt period available. The fixed-interval timer exception occurs on 0 to 1 transitions of a selected bit from the time base (see Section 9.2.2, “Timer Control Register (TCR)”).

The fixed-interval timer exception is logged by TSR[FIS]. A fixed-interval timer interrupt will occur if TCR[FIE] and MSR[EE] are enabled. See Section 5.7.10, “Fixed-Interval Timer Interrupt,” for details of register behavior caused by the fixed-interval timer interrupt.

The interrupt handler must reset TSR[FIS] via software, in order to avoid another fixed-interval timer interrupt once MSR[EE] is re-enabled (assuming TCR[FIE] is not cleared instead). This is done by writing a word to the timer status register using **mtspr** with a 1 in the bit corresponding to TSR[FIS] (and any other bits that are to be cleared) and 0 in

all other bits. The write-data to the TSR is not direct data, but is a mask. Writing a 1 causes the bit to be cleared; writing a 0 has no effect.

A fixed-interval timer interrupt also occurs if the selected time base bit changes from 0 to 1 due to an **mtspr** that writes a 1 to a bit that was previously a 0. The fixed-interval timer interrupt is described in Section 5.7.10, “Fixed-Interval Timer Interrupt.”

9.5 Watchdog Timer

The watchdog timer is a facility intended to aid system recovery from faulty software or hardware. Watchdog time-outs occur on 0 to 1 transitions of selected bits from the time base (see Section 9.2.2, “Timer Control Register (TCR)”).

When a watchdog timer time-out occurs while watchdog timer interrupt status is clear (TSR[WIS] = 0) and the next watchdog time-out is enabled (TSR[ENW] = 1), a watchdog timer exception is generated and logged by setting TSR[WIS]. This is referred to as a watchdog timer first time out. A watchdog timer interrupt occurs if enabled by TCR[WIE] and MSR[CE]. See Section 5.7.11, “Watchdog Timer Interrupt,” for details of register behavior caused by the watchdog timer interrupt.

To avoid another watchdog timer interrupt once MSR[CE] is re-enabled (assuming TCR[WIE] is not cleared instead), the interrupt handler must reset TSR[WIS] through software. This is done by writing a word to TSR using **mtspr** with a 1 in the bit corresponding to TSR[WIS] (and any other bits that are to be cleared) and a 0 in all other bits. The write-data to the TSR is not direct data, but is a mask. A 1 causes the bit to be cleared; a 0 has no effect.

Note that a watchdog timer exception also occurs if the selected time base bit transitions from 0 to 1 due to an **mtspr** that writes a 1 to the bit when its previous value was 0.

When a watchdog timer time-out occurs while TSR[WIS] = 1 and TSR[ENW] = 1, a processor reset occurs if it is enabled by a non-zero value of the watchdog reset control field in the timer control register (TCR[WRC]). This is referred to as a watchdog timer second time out. The assumption is that TSR[WIS] was not cleared because the processor was unable to execute the watchdog timer interrupt handler, leaving reset as the only available means to restart the system. Note that once TCR[WRC] has been set to a non-zero value, it cannot be reset by software; this feature prevents errant software from disabling the watchdog timer reset capability.

Figure 9-8 gives a more complete view of watchdog timer behavior.

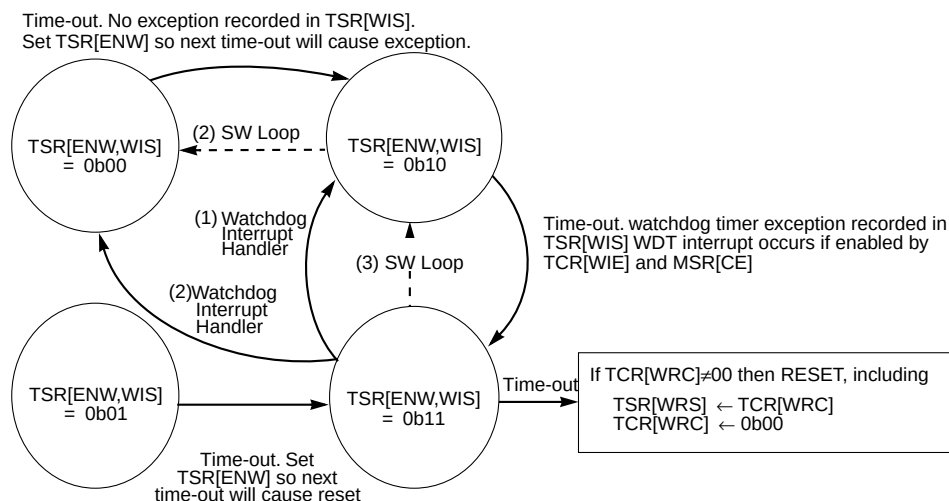


Figure 9-8. Watchdog State Machine

The numbers in parentheses in Figure 9-8 refer to the following discussion of modes of operation:

1. Always take the watchdog timer interrupt when pending, and never attempt to prevent its occurrence. In this mode, the watchdog timer interrupt caused by a first time-out is used to clear `TSR[WIS]` so a second time-out never occurs. `TSR[ENW]` is not cleared, thereby allowing the next time-out to cause another interrupt.
2. Always take the watchdog timer interrupt when pending, but avoid when possible. In this mode, a recurring code loop of reliable duration (or perhaps a periodic interrupt handler such as the fixed-interval timer interrupt handler) is used to repeatedly clear `TSR[ENW]` such that a first time-out exception is avoided, and thus no watchdog timer interrupt occurs. Once `TSR[ENW]` has been cleared, software has between one and two full watchdog periods before a watchdog exception is posted in `TSR[WIS]`. If this occurs before the software can clear `TSR[ENW]` again, a watchdog timer interrupt will occur. In this case, the watchdog timer interrupt handler then clears both `TSR[ENW]` and `TSR[WIS]`, in an attempt to avoid the next watchdog timer interrupt.
3. Never take the watchdog timer interrupt. In this mode, watchdog timer interrupts are disabled (via `TCR[WIE]` is cleared), and the system depends upon a recurring code loop of reliable duration (or perhaps a periodic interrupt handler such as the fixed-interval timer interrupt handler) to repeatedly clear `TSR[WIS]` such that a second time-out is avoided, and thus no reset occurs. `TSR[ENW]` is not cleared, thereby allowing the next time-out to set `TSR[WIS]` again. The recurring code loop must have a period that is less than one watchdog timer period to guarantee that a watchdog timer reset does not occur.

Table 9-3 describes the watchdog timer state machine and watchdog timer controls.

Table 9-3. Watchdog Timer Controls

Enable Next Watchdog Timer (TSR[ENW])	Timer Status (TSR[WIS])	Action when Timer Interval Expires
0	0	Set enable next watchdog timer (TSR[ENW] is set).
0	1	Set enable next watchdog timer (TSR[ENW] is set).
1	0	Set watchdog timer interrupt status bit (TSR[WIS] is set). If the watchdog timer interrupt is enabled (TCR[WIE] is set and MSR[CE] is set), then interrupt.
1	1	Cause watchdog timer reset action specified by TCR[WRC]. Reset will copy prereset TCR[WRC] into TSR[WRS], then clear TCR[WRC].

The controls described in Table 9-3 imply three different modes of operation that a programmer might select for the watchdog timer. Each of these modes assumes that TCR[WRC] has been set to allow processor reset by the watchdog facility:

9.6 Freezing the Timer Facilities

The debug mechanism provides a means of temporarily freezing the timers when a debug event occurs. Specifically, the time base and decrementer can be frozen and prevented from incrementing/decrementing, respectively, whenever a debug event is set in the DBSR. This allows a debugger to simulate the appearance of real time, even though the application has been temporarily halted to service the debug event. See the description of DBCR0[FT] (freeze timers on debug event) in Section 2.13.1.1, “Debug Control Register 0 (DBCR0).”

Chapter 10

Auxiliary Processing Units (APUs)

This chapter describes the APUs implemented on the e500, which are as follows:

- Section 10.1, “Instruction Select APU”
- Section 10.2, “Performance Monitor APU”
- Section 10.3, “Signal Processing Engine APU (SPE APU)”
- Section 10.4, “Branch Target Buffer (BTB) Locking APU”
- Section 10.5, “Cache Block Lock and Unlock APU”
- Section 10.6, “Machine Check APU”

10.1 Instruction Select APU

Control code, which is characterized by unpredictable short branches, is common in embedded applications. When mispredicted, these branches cause long pipeline delays. The instruction select APU consists of a single instruction (**isel**), which is a conditional register move that helps eliminate some of these branches. The **isel** instruction works as follows:

```
if crB then
    rD = rA
else
    rD = rB
```

The **isel** instruction allows more efficient implementation of a condition sequence such as the one in the following generic example:

```
int16 global1,..., global37,...;
....
void procedure17(int16 parm) {
    if (global1 == 27) {
        global37 = parm + 17;
    }
    else {
        global37 = parm - 17;
    }
}
```

10.1.1 Instruction Select APU Programming Model

The instruction select APU includes only the **isel** instruction, described in the EREF. It accesses the GPRs and condition register and does not implement additional registers or interrupt resources.

10.1.2 Using isel to Improve Conditional Branch Performance

The Instruction Select instruction, **isel**, can be used to more efficiently handle short conditional branch segments. **isel** has two source registers and one destination register. Under the control of a specified condition code bit, it copies one or the other source operand to the destination.

Table 10-1 shows a coding example with and without the **isel** instruction.

Table 10-1. Recoding with isel

Code Sequence without isel	Code Sequence with isel
<pre>cmpi cr3, r17, 27; bne cr3, NotEqual; addi r15, r17, 17; jmp Assign; NotEqual: addi r15, r17, -17; Assign: stw r15, (rGlobals + g37);</pre>	<pre>cmpi cr3, r17, 27; addi r15, r17, 17; addi r16, r17, -17; isel r15, r15, r16, cr3.eq stw r15, (rGlobals + g37);</pre>

The sequence without **isel** turns conditional branches into a code sequence that sets a condition code according to the results of a computation. It uses a conditional branch to choose a target sequence, but needs an unconditional branch for the IF clause. The conditional branch is often hard to predict, the code sequences are generally small, and the resulting throughput is typically low.

The sequence using **isel** does the following:

- Sets a condition code according to the results of a comparison
- Has code that executes both the IF and the ELSE segments
- Has a final statement that copies the results of one of the segments to the desired destination register
- Works well for small code segments and for unpredictable branches
- Can reduce code size

10.2 Performance Monitor APU

The e500 core complex architecture defines a performance monitor facility that provides the ability to monitor and count predefined events such as processor clocks, misses in the

instruction cache or data cache, types of instructions decoded, or mispredicted branches. The count of such events can be used to trigger the performance monitor exception.

The performance monitor APU is described in detail in Chapter 7, “Performance Monitor.”

The performance monitor can be used to do the following:

- Improve system performance by monitoring software execution and then recoding algorithms for more efficiency. For example, memory hierarchy behavior can be monitored and analyzed to optimize task scheduling or data distribution algorithms.
- Characterize processors in environments not easily characterized by a benchmark or trace
- Help system developers bring up and debug their systems

The performance monitor uses the following resources:

- The performance monitor mark bit in the MSR (MSR[PMM]). This bit controls which programs are monitored.
- The move to/from PMR instructions, **mtpmr** and **mfpmr**.
- The external input pin, PM_EVENT.
- Performance monitor registers (PMRs):
 - The performance monitor counter registers (PMC0–PMC3) are 32-bit counters used to count software-selectable events. Each counter counts up to 64 reference events and up to 64 counter-specific events. UPMC0–UPMC3 provide user-level read access to these registers. Reference events are those that should be applicable to most microprocessor microarchitectures and be of general value. They are identified in Table 7-2.
 - The performance monitor global control register (PMGC0) controls the counting of performance monitor events. It takes priority over all other performance monitor control registers. UPMGC0 provides user-level read access to this register.
 - The performance monitor local control registers (PMLCa0–PMLCa3, PMLCb0–PMLCb3) control each individual performance monitor counter. Each counter has a corresponding PMLCa and PMLCb register. UPMLCa0–UPMLCa3 and UPMLCb0–UPMLCb3 provide user-level read access to PMLCa0–PMLCa3, PMLCb0–PMLCb3).
- The performance monitor interrupt follows the Book E interrupt model and is assigned to interrupt vector offset register 35 (IVOR35). Its priority is less than the fixed-interval interrupt and greater than the decremter interrupt.

The e500 core complex implements a performance monitor as an APU. Software communication with the performance monitor APU is achieved through performance monitor registers (PMRs) rather than SPRs. The PMRs are used for enabling conditions that can trigger a APU-defined performance monitor interrupt.

10.3 Signal Processing Engine APU (SPE APU)

This section describes the SPE APU programming model, exceptions, and functions.

10.3.1 SPE APU Programming Model

The e500 core complex provides a register file with thirty-two 64-bit general-purpose registers (GPRs). The AIM definition of the PowerPC architecture and 32-bit Book E instructions operate on bits 32–63 of 64-bit GPRs. New SPE vector instructions are defined that view the 64-bit register as being composed of a vector of two 32-bit elements (some of the instructions also read or write 16-bit elements). These new instructions can also perform scalar DSP operations by ignoring the results of the upper 32-bit half of the register file. Some scalar instructions are defined for DSP that produce a 64-bit scalar result. Vector floating-point instructions operate on a vector of two 32-bit, single-precision floating-point numbers resident in the 64-bit registers. Single-precision vector floating-point instructions do not have a separate register file; there is a single shared register file for all instructions. The SPE APU and Book E instructions issue from a single instruction stream.

There is no record form of SPE APU instructions. Vector compare instructions store the result of the comparison into the condition register (CR). The meaning of the CR bits are now overloaded for the vector operations. Vector compare instructions specify a CR field and two source registers and also the type of compare: greater than, less than, or equal. Two bits in the CR field are written with the result of the vector compare, one for each element. The remaining two bits are undefined. The two defined bits could be used either by a vector select instruction or the PowerPC UISA branch instruction.

A partially visible accumulator register is architected for the integer/fractional SPE APU multiply accumulate forms of instructions. Its usage is described in Section 2.14.2, “Accumulator (ACC).”

10.3.1.1 SPE APU Register Model

SPE APU instructions use GPRs for source and destination operands. SPE vector instructions are the only e500 instructions that use the upper 32 bits in the GPRs. SPE scalar floating-point instructions use only the lower 32 bits. Note that a value in the upper word of a 64-bit GPR is not altered by a 32-bit instruction. So, the value in the upper half of the GPR can be read from the GPR even after a scalar value has been written to the lower half.

The SPE defines the following additional register resources:

- SPE-specific bits in the MSR and ESR. These are described in Section 10.3.1.1.1, “Allocated Bits in Book E–Defined Registers.”
- Signal processing and embedded floating-point status and control register (SPEFSCR). This is described in Section 10.3.1.1.2, “Signal Processing and Embedded Floating-Point Status and Control Register (SPEFSCR).”

- An accumulator, described in Section 2.14.2, “Accumulator (ACC).”
- SPE-specific interrupt vector offset registers (IVORs) used for locating SPE-specific interrupts. See Section 10.3.1.1.4, “SPE Interrupt Vector Offset Registers.”

The SPE defines the signal processing and embedded floating-point status and control register (SPEFSCR), which is described in Section 2.14.1, “Signal Processing and Embedded Floating-Point Status and Control Register (SPEFSCR).”

10.3.1.1.1 Allocated Bits in Book E–Defined Registers

The following SPE-specific bits are defined in the MSR and ESR.

- Vector available bit—MSR[6] is defined as the vector available bit (SPE). If this bit is clear and software attempts to execute any of the SPE instructions, the SPE APU unavailable exception is taken. If this bit is set, software can execute any SPE instructions.
- Vector exception bit—ESR[24] is defined as the vector exception bit (SPE). This bit is set whenever the processor takes an exception related to the execution of the SPE APU instructions.

10.3.1.1.2 Signal Processing and Embedded Floating-Point Status and Control Register (SPEFSCR)

Embedded vector and floating-point exception conditions are signaled through the SPEFSCR, which is described in Section 2.14.1, “Signal Processing and Embedded Floating-Point Status and Control Register (SPEFSCR).”

Floating-point conditions are described as follows:

- Embedded floating-point invalid operation. The embedded floating-point invalid operation is signaled in the SPEFSCR[FINVH,FINV] when the input to a floating-point instruction is an Infinity, Denorm, or NaN. Additionally, FINVH and FINV are signaled on a floating-point divide if both the dividend and divisor are zero.
- Embedded floating-point underflow. The embedded floating-point underflow is signaled in the following conditions: ($rD = \pm 0$)
 - Floating-point multiply:

$$\text{Exponent biased intermediate} = E_a + E_b - \text{bias}$$

$$E_i < E_{\min}(0) \rightarrow \text{FUNF/FUNFH}$$
 - Floating-point divide:

$$\text{Exponent biased intermediate} = E_a - E_b + \text{bias}$$

$$E_i \leq E_{\min}(0) \rightarrow \text{FUNF/FUNFH}$$
 - Floating-point add/subtract:

Exponent biased intermediate = $\text{Max}(E_a, E_b)$

E_n = Exponent biased intermediate—normalization factor

Normalization factor = amount of shift to normalize MantissaA to MantissaB

$E_n < E_{\text{min}}(0) \rightarrow \text{FUNF/FUNFH}$

- Embedded floating-point overflow: The embedded floating-point overflow is signaled in the following conditions: $[\text{RD} = \pm \text{FPMAX}]$

— Floating-point multiply:

Exponent biased intermediate = $E_a + E_b - \text{bias}$

$E_i \geq E_{\text{max}}(255) \rightarrow \text{FOVF/FOVFH}$

— Floating-point divide:

Exponent biased intermediate = $E_a - E_b + \text{bias}$

$E_i > E_{\text{max}}(255) \rightarrow \text{FOVF/FOVFH}$

— Floating-point add/subtract:

Exponent biased intermediate = $\text{Max}(E_a, E_b)$

$E_i \geq E_{\text{max}}(255) \rightarrow \text{FOVF/FOVFH}$

— Floating-point convert to signed integer:

$E_b \geq 158 (\text{bias} + 31) \rightarrow \text{FOVF/FOVFH}$

— Floating-point convert to unsigned integer:

$E_b \geq 159 (\text{bias} + 32) \rightarrow \text{FOVF/FOVFH}$

$\text{SignB} = 1 \rightarrow \text{FOVF/FOVFH}$

— Floating-point convert to signed fractional:

$E_b \geq 127 (\text{bias}) \rightarrow \text{FOVF/FOVFH}$

— Floating-point convert to unsigned fractional:

$E_b \geq 127 (\text{bias}) \rightarrow \text{FOVF/FOVFH}$

$\text{SignB} = 1 \rightarrow \text{FOVF/FOVFH}$

NOTE

If an invalid operation error occurs, overflow or underflow conditions are not signalled. However, if the exception is not enabled for invalid operation errors, invalid inputs are treated as underflow or overflow values and **rD** is updated as if an underflow or an overflow were signalled.

10.3.1.1.3 Accumulator

The 64-bit accumulator register holds the results of the multiply accumulate (MAC) forms of SPE integer instructions. The accumulator allows back-to-back execution of dependent

MAC instructions, something that is found in the inner loops of DSP code such as FIR filters. The accumulator is partially visible to the programmer in that its results do not have to be explicitly read to use them. Instead, they are always copied into a 64-bit destination GPR specified as part of the instruction. The accumulator however has to be explicitly cleared when starting a new MAC loop. Based upon the type of instruction, an accumulator can hold either a single 64-bit value or a vector of two 32-bit elements.

Figure 10-1 shows an example of a MAC instruction, **evmhossfaaw rD,rA,rB**. In this instruction, the least significant 16 bits of **rA** and **rB** are multiplied for both elements of the vector and the result is shifted left one bit and added to the accumulator and the result is possibly saturated to 32 bits in case of overflow. The final result is placed both in the accumulator and also in **rD**. Thus, the result of this instruction can be used by accessing **rD**.

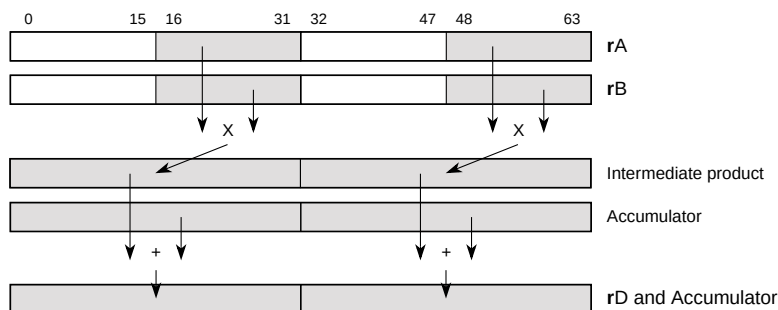


Figure 10-1. Odd Form of Vector Half-Word Multiply (evmhossfaaw)

To read the accumulator contents into a **rD**, a multiply-accumulate instruction where one of its operands is a zero should be used, as the following sequence shows:

```
evxor rD,rD,rD           // Zero the contents of RD, not necessary if
                          // a zero is available in some register.
evmwumiaa rD,rD,rD       // Multiply 0 with 0, add the 0 result to
                          // accumulator and store back the value in acc and RD
```

How the accumulator is handled for subroutine calls and context switching is described as follows:

- **Subroutine calls**—Because the result of a MAC instruction is always available in a GPR, the accumulator does not have to be explicitly saved by either the caller or the callee. A callee executing any MAC instructions must explicitly clear the accumulator before the MAC. After the return from the subroutine call, the caller must reload the content of the accumulator from the GPR to which it was copied. This is because the caller cannot know if the callee modified the accumulator.
- **Context switch**—When a context switch occurs, the OS processor or the TMU must explicitly save the accumulator as part of the context of the swapped-out task and then explicitly load the accumulator from the context of the new task that is being

swapped in. When the old task is being scheduled, its accumulator must be restored before restarting the task.

10.3.1.1.4 SPE Interrupt Vector Offset Registers

Three interrupt vector offset registers, IVOR32, IVOR33, and IVOR34, are used by the exception mode. These supervisor-level SPRs are accessed as SPR 528, 529, and 530, respectively. Attempting to access them in user mode causes a privileged instruction exception.

The SPE implements the following interrupt vector offset registers (IVORs):

- SPE APU interrupt vector offset register (IVOR32), implemented as SPR 528
- Vector floating-point data interrupt vector offset register (IVOR33), implemented as SPR 529
- Vector floating-point round interrupt vector offset register (IVOR34), implemented as SPR 530

IVOR registers are described in Section 2.7.1.7, “Interrupt Vector Offset Registers (IVORs).”

10.3.1.2 SPE APU Instruction Model

The SPE implements the following types of instructions:

- SPE vector integer instructions, which operate on 64-bit GPRs. Individual instructions are described in Section 3.11.1.1, “SPE APU Vector Instructions.”
- SPE vector load and store instructions, which operate on 64-bit GPRs. Individual instructions are described in Section 3.11.1.1, “SPE APU Vector Instructions.”
- SPE floating-point instructions, described in Section 3.11.1.2, “SPE APU Floating-Point Instructions.” These include the following:
 - A set of vector floating-point instructions that perform operations on two-element single-precision operands that reside in 64-bit GPRs
 - A set of scalar floating-point instructions that perform operations on single-precision operands that reside in bits 32–63 of the 64-bit GPRs. These instructions are also referred to as the SPFPUs.

10.3.1.2.1 PowerPC UISA and Book E Instructions

The e500 core complex implements the 32-bit forms of the Book E instructions. All 32-bit PowerPC UISA and Book E instructions operate upon the lower half of the 64-bit register. These instructions do not affect the upper half of the register.

Note that there are signed and unsigned forms and optional fractional forms. For all of these instructions, the fractional form does not apply to unsigned forms because integer and fractional forms are identical for unsigned operands.

10.3.1.2.2 IEEE 754 Compliance

The architecture specifies that the processor implement a single-precision floating-point system as defined in ANSI/IEEE Standard 754-1985 but requires software support in order to conform fully with the standard. Whenever an input operand of the floating-point instruction has data values that are + infinity, - infinity, or a NaN or when the result of an operation may produce an overflow or an underflow, an exception is taken and the exception handler is responsible for delivering IEEE 754-compliant behavior.

Two rounding modes, round to nearest and round to zero, are implemented in the e500 core complex. The remaining two rounding modes, round to +infinity and round to -infinity, are supported in software. When SSR[EVRN] indicates round to +infinity or round to -infinity, an exception is taken and the result that is rounded to zero is written in the register and the guard and sticky bits of the operation are reported. Software should use this information to deliver the correctly rounded result.

10.3.1.2.3 Signal-Processing Floating-Point (SPFPU) Operations

The e500 core complex implements SPE floating-point instructions that operate upon the contents of a 64-bit register that is a vector of two single-precision data elements. The floating-point unit shares the same register file as the integer unit. There is no separate floating-point register file. As is the case for most SPE operations, the floating-point instructions can also be used to perform scalar floating-point operations by ignoring the values in the upper half of the register file. The PowerPC UISA and Book E floating-point instructions are not implemented in the e500 core complex.

10.3.2 SPE APU Exceptions

The SPE APU defines the following exceptions:

- SPE APU unavailable exception
- Vector floating-point data exception
- Vector floating-point round exception
- Vector alignment exception

For more information, see Section 5.7.15, “e500-Specific Exceptions.”

10.3.2.1 SPE APU Unavailable Exception

The SPE APU unavailable exception is taken if MSR[SPE] is cleared and a SPE APU instruction is executed. When the SPE APU unavailable exception occurs, the processor suppresses execution of the instruction causing the exception. SRR0, SRR1, MSR, and ESR are modified as follows:

- SRR0 is set to the effective address of the instruction causing the exception.
- SRR1 is set to the contents of the MSR at the time of the exception.
- MSR[CE,ME,DE] are unchanged. All other bits are cleared.
- ESR[SPE] (bit 24) is set. All other ESR bits are cleared.

Instruction execution resumes at address IVPR[32–47] || IVOR32[48–59] || 0b0000.

10.3.2.2 Vector Floating-Point Data Exception

The vector floating-point data exception is taken if SPEFSCR[EVFEH] is set and any of the SPEFSCR[FIEH], SPEFSCR[FOVH], SPEFSCR[FUNH] bits are set, or, SPEFSCR[EVFEL] is set and one or more of the SPEFSCR[FIEL], SPEFSCR[FOVL], SPEFSCR[FUNL] bits are set. When the vector floating-point data exception occurs, the processor suppresses the execution of the instruction causing the exception. SRR0, SRR1, MSR, ESR, and SPEFSCR are modified as follows:

- SRR0 is set to the effective address of the instruction causing the exception.
- SRR1 is set to the contents of the MSR at the time of the exception.
- MSR[CE,ME,DE] are unchanged. All other bits are cleared.
- ESR[SPE] (bit 24) is set. All other ESR bits are cleared.
- One or more SPEFSCR status bits are set to indicate the type of exception. The affected bits are FSEH, FIEH, FOVH, FUNH, FSEL, FIEL, FOVL, and FUNL.

Instruction execution resumes at address IVPR[32–47] || IVOR33[48–59] || 0b0000.

10.3.2.3 Vector Floating-Point Round Exception

The vector floating-point round exception is taken if at least one of the following conditions are met:

- The floating-point round mode indicated by SPEFSCR[EVRN] is either round to nearest or round to zero, SPEFSCR[EVXE] is set, and the rounded result of the operation is not exact in any element of the vector.
- The floating-point round mode indicated by SPEFSCR[EVRN] is either round to +inf or round to -inf.

When a vector floating-point round exception occurs, the unrounded (truncated) result is placed in the target registers. SRR0, SRR1, MSR, ESR, and SPEFSCR are modified as follows:

- SRR0 is set to the effective address of the instruction following the instruction causing the exception.
- SRR1 is set to the contents of the MSR at the time of the exception.
- MSR[CE,ME,DE] are unchanged. All other MSR bits are cleared.
- ESR[SPE] (bit 24) is set. All other ESR bits are cleared.
- SPEFSCR[GH,GH,XH,XL] are set appropriately.

Instruction execution resumes at address IVPR[32–47] || IVOR34[48–59] || 0b0000.

10.3.2.4 Vector Alignment Exception

The vector alignment exception is taken if the real address of the following instructions is not aligned to the 64-bit boundary: **evldd**, **evlddx**, **evldw**, **evldwx**, **evldh**, **evldhx**, **evstdd**, **evstddx**, **evstdw**, **evstdwx**, **evstdh**, and **evstdhx**. When the alignment exception occurs, the processor suppresses the execution of the instruction causing the alignment exception. SRR0, SRR1, MSR, ESR, and DEAR are modified as follows:

- SRR0 is set to the effective address of the instruction causing the alignment exception.
- SRR1 is set to the contents of the MSR at the time of the exception.
- MSR[CE,ME,DE] are unchanged. All other bits are cleared.
- ESR[SPE] (bit 24) and ESR[AP] are set. ESR[ST] is set only if the instruction causing the exception is a store and is cleared for a load. All other bits are cleared.
- DEAR is modified as described Section 5.3, “Interrupt Registers.”

Instruction execution resumes at address IVPR[32–47] || IVOR5[48–59] || 0b0000.

10.3.2.5 Exception Priorities

The following list shows the order in which simultaneous exceptions are taken:

1. SPE APU unavailable exception
2. Vector floating-point data exception
3. Vector floating-point round exception

10.3.3 SPE APU Functions

```
// Determine if fp value is a NaN or Infinity
IsNaNNorInfinity(fp)
return (fpexp == 255)

// Determine if fp value is denormalized
```

Signal Processing Engine APU (SPE APU)

```
IsaDenorm(fp)
return ((fpexp == 0) & (fpfrac != 0))

// Signal a floating-point error in the SPEFSCR
SignalFPError(upper_lower, bits)
if (upper_lower == UPPER) then
    bits = bits << 15
SPEFSCR = SPEFSCR | bits

// Convert 32 bit floating point to integer/factional
//   signed = SIGN or UNSIGN
//   upper_lower = UPPER or LOWER
//   round = ROUND or TRUNC
//   fractional = F or I
CnvtFP32ToISat(fp, signed, upper_lower, round, fractional)

if (IsaNaNorInfinity(fp)) then // SNaN, QNaN, +-INF
    SignalFPError(upper_lower, FSE | FIE)
    if (signed == SIGN) then
        if (fpsign == 1) then
            return 0x80000000
        else
            return 0x7fffffff
    else
        if (fpsign == 1) then
            return 0x00000000
        else
            return 0xffffffff

if (IsaDenorm(fp)) then
    SignalFPError(upper_lower, FSE | FIE)
    return 0x00000000 // regardless of sign
if (signed == UNSIGN & fpsign == 1) then
    SignalFPError(upper_lower, SOV | OV) // overflow
    return 0x00000000

if (fp == 0) then
    return 0

if (fractional == I) then // convert to integer
    max_exp = 158
    shift = 158 - fpexp
    if (signed == SIGN) then
        max_exp = max_exp - 1
else // fractional conversion
    max_exp = 126
    shift = 126 - fpexp
    if (signed == SIGN) then
        shift = shift + 1

if (fpexp > max_exp) then
    SignalFPError(upper_lower, SOV | OV) // overflow
    if (signed == SIGN) then
        if (fpsign == 1) then
            return 0x80000000
        else
            return 0x7fffffff
    else
        return 0xffffffff

result = 0b1 || fpfrac || 0b00000000 // add U to frac
guard = sticky = 0

for (n = 0; n < shift; n = n + 1) do
    sticky = sticky | guard
    guard = result & 0x00000001
    result = result > 1
```

```

// Report sticky and guard bits
if (upper_lower == UPPER) then
    FGH = guard
    FXH = sticky
else
    FGL = guard
    FXL = sticky

// Round the result (only need to make changes for
// round to nearest since round to zero is truncation
// and infinity rounding must be handled by software)
if (round == ROUND) then
    if (SPEFSCR[EVRN] == 0b00) then
        if (guard) then
            if (sticky | (result & 0x00000001)) then
                result = result + 1

if (signed == SIGN) then
    if (fpsign == 1) then
        result = ~result + 1

return result

// Handle saturation
SATURATE(flag, sat_val, val)
if (flag) then
    return sat_val
else
    return val

// Handled signed saturate
SATURATE_ACC(flag, sign, negative_sat_val, positive_sat_val, val)
if (flag) then
    if (sign) then
        return negative_sat_val
    else
        return positive_sat_val
else
    return val

```

10.4 Branch Target Buffer (BTB) Locking APU

The core complex provides a 512-entry branch target buffer (BTB) for efficient processing of branch instructions. The BTB is a branch target address cache, organized as 128 rows with four-way set associativity, that holds the address and target instruction of the 512 most-recently taken branches, each with a 2-bit, dynamically updated branch history table that indicates four levels of likelihood that the branch will be taken (strongly taken, taken, not taken, strongly not taken). The BTB provide quick access to branch targets and history bits that allow efficient branch prediction.

The core complex also provides support for locking and unlocking BTB entries for deterministic branch behavior. In particular, the BTB locking APU gives the user the ability to lock, unlock, and invalidate entries in the BTB.

10.4.1 BTB Locking APU Programming Model

The branch target buffer APU defines additional instructions and register resources, which are described in the following sections. It does not define additional interrupts. See Chapter 4, “Execution Timing.”

10.4.1.1 Branch Target Buffer Instructions

Table 10-2 lists the BTB instructions, which are described in detail in the EREF.

Table 10-2. Branch Target Buffer (BTB) Instructions

Name	Mnemonic	Syntax
Branch Buffer Load Entry and Lock Set	bblels	—
Branch Buffer Entry Lock Reset	bbelr	—

10.4.1.2 Branch Target Buffer Registers

The BTB locking APU register model includes the following register resources for enabling the locking and unlocking of BTB entries:

- Branch unit control and status register (BUCSR)—SPR 1013. This register has bits that are used to enable or disable BTB locking and to control unlocking, invalidation, and overlocking of BTB entries. See Section 2.9, “Branch Target Buffer (BTB) Registers.”
- Branch buffer entry address register (BBEAR)—SPR 512. This register holds the address of a BTB entry. See Section 2.9.1, “Branch Buffer Entry Address Register (BBEAR).”
- Branch buffer target address register (BBTAR)—SPR 513. This register includes branch target address bits and a field that allows the programmer to specify whether a branch should be predicted as taken or not taken. See Section 2.9.2, “Branch Buffer Target Address Register (BBTAR).”
- MSR[UBLE], the user branch locking enable bit, determines whether user mode programs can lock or unlock BTB entries. See Section 2.5.1, “Machine State Register (MSR).”

10.5 Cache Block Lock and Unlock APU

The cache block lock and unlock APU allows instructions and data to be locked into their respective caches on a cache line basis. Locking is performed by a set of touch and lock set instructions. This functionality can be enabled for user mode by setting MSR[UCLE].

Cache locking requires the appropriate access to the address. The APU also provides resources for detecting and handling overlocking conditions.

Cache locking exception conditions are defined for data storage interrupt.

10.5.1 Cache Block Lock and Unlock APU Programming Model

This section describes instruction, register, and interrupt models.

10.5.1.1 Cache Block Lock and Unlock APU Instruction Model

This section describes the instructions in the cache block lock and unlock APU, which consists of the instructions described in Table 3-42. Full descriptions of these instructions can be found in the EREF. The CT operand affects whether the instruction affects the L1 or L2 cache.

- **Data Cache Block Lock Clear (dcblic)**—If CT=0 and the line is in the data cache, the data cache lock bit for that line is cleared, making it eligible for replacement. If CT=1 and the line is in the L2 cache, the lock bit for that line is cleared, making it eligible for replacement.
- **Data Cache Block Touch and Lock Set (dcbtlls)**—If CT=0, the line is loaded and locked into the L1 data cache. If CT=1, the line is loaded into the unified L2 cache and the line is locked into the L2 cache. If CT=1 and the block is in the L2 cache, **dcbtlls** marks the block so it is not a candidate replacement.
- **Data Cache Block Touch for Store and Lock Set (dcbtstls)** If CT = 0, the e500 core fetches the block containing the byte addressed by EA into the data cache. The access of the byte probably has stringent latency requirements, so after the block containing the byte is fetched, it is locked. If CT = 0 and the block is in the data cache, **dcbtstls[e]** marks the block locked so it is no longer eligible for replacement. If CT=1 and the block is in the L2 cache, **dcbtstls** marks the block such that it should not be selected for replacement.
If CT is not 0 or 1, **dcbtstls** is no-oped. In the L1 data cache, the e500 implements a lock bit for every index and way, allowing a line locking granularity.
- **Instruction Cache Block Lock Clear (icblic)** If CT=0 and the line is in the instruction cache, the instruction cache lock bit for that line is cleared, making that line eligible for replacement. If CT=1, and the line is in the L2 cache, the lock bit for that line is cleared in the L2 cache, making it eligible for replacement. If CT is not 0 or 1, the **icblic** is NOPed.
- **Instruction Cache Block Touch and Lock Set (icbtlls)** If CT=0, the line is loaded and locked into the L1 data cache. If CT=1, the line is loaded into the unified L2 cache, and the line is locked into the L2 cache. If CT=1 and the block already exists in the L2 cache, **icbtlls** marks the block such that it should not be selected for replacement. If CT is not 0 or 1, **icbtlls** is NOPed.

Note that the touch and lock set instructions (**icbtl**s, **dcbtl**s, and **dcbtstl**s) are always executed and are not treated as hints. When one of these instructions is performed to an index and the way cannot be locked, L1CSR1[ICUL] or L1CSR0[DCUL] is set to indicate an unable-to-lock condition. This occurs if the instruction must be NOPed.

10.5.1.2 Cache Locking APU Register Model

The cache locking APU implements fields in the L1 cache control and status registers (L1CSR0 and L1CSR1). Status bits indicate whether an internally or externally generated **dcbi** snoop invalidated a locked line, whether a lock set instruction was effective in the cache, or whether a lock overflow condition was encountered in the instruction or data cache. The cache lock bits flash clear field allows flash clearing of lock bits of all cache entries. For more information, see Section 2.11, “L1 Cache Configuration Registers.”

The DLK and ILK bits in the exception syndrome register [ESR] indicate the occurrence of a data or instruction locking exception condition, which generates a data storage interrupt.

10.5.1.3 Cache Locking APU Interrupt Model

The DLK and ILK bits in the exception syndrome register [ESR] indicate the occurrence of a data or instruction locking exception condition, which generates a data storage interrupt. See Section 5.7.3, “Data Storage Interrupt.”

10.6 Machine Check APU

The machine check APU defines features for the machine check interrupt in addition to those defined by the PowerPC architecture and the Book E version of the PowerPC architecture. The machine check APU includes an enhanced definition the machine check interrupt type similar to the Book E-defined critical interrupt.

10.6.1 Machine Check APU Programming Model

The APU defines dedicated save and restore SPRs, MSRR0 and MSRR1, so a machine check interrupt does not affect the CSRR0, CSRR1, or ESR registers as defined by the Book E architecture.

The APU also defines a separate Return from Machine Check Interrupt instruction, **rfmci**, that restores context from the MSRR0 and MSRR1 when the machine check interrupt handler completes.

10.6.1.1 Machine Check APU Register Model

The machine check APU defines different register resources than the Book E definition, which are as follows:

- Machine-check save/restore register 0 (MCSRR0)—SPR 570. This register holds the instruction where fetching begins after the **rfmci** instruction executes, typically at the end of the machine check interrupt handler. This register is described in Section 2.7.2.1, “Machine Check Save/Restore Register 0 (MCSRR0).”
- Machine-check save/restore register 1 (MCSRR1)—SPR 571. This register holds the machine state copied to the MSR when a machine check interrupt occurs. The value of this register is restored to the MSR when an **rfmci** instruction executes, typically at the end of the machine check interrupt handler. This register is described in Section 2.7.2.2, “Machine Check Save/Restore Register 1 (MCSRR1).”
- Machine check syndrome register (MCSR)—SPR 572. The MCSR has fields that identify the causes for a machine check interrupt along with an indication of whether the processor can recover from the machine check interrupt. This register is described in Section 2.7.2.4, “Machine Check Syndrome Register (MCSR).”
- Machine check address register (MCAR)—SPR 573. The MCAR holds the address of the data or instruction that caused a machine check interrupt. This register is described in Section 2.7.2.3, “Machine Check Address Register (MCAR).”

Note, however, that the MSR[ME] bit, defined by the original PowerPC architecture is also used in Book E and in the machine check APU to enable the machine check interrupt.

10.6.1.2 Machine Check APU Instruction Model

The Return from Machine Check instruction, **rfmci**, is context-synchronizing; it works its way to the final execute stage, updates architected registers, and redirects instruction flow. When **rfmci** executes, data is restored from MCSRR0 and MCSRR1. The **rfi** and **rfci** instructions do not affect MCSRR0 and MCSRR1. This instruction is described in Chapter 3, “Instruction Model.”

10.6.1.3 Machine Check Interrupt

The machine check APU is consistent with the machine check exception as defined in Book E with the following differences:

- Machine check is no longer a critical interrupt but uses MCSRR0 and MCSRR1 for saving the return address and the MSR in case the machine check is recoverable.
- Return from Machine Check Interrupt instruction (**rfmci**) is implemented to support the return to the address saved in MCSRR0.
- The machine check syndrome register, MCSR, is used (instead of ESR) to log the cause of the machine check.

The behavior of the machine check interrupt is described in Section 5.7.2, “Machine Check Interrupt.”

Chapter 11

Cache and MMU Background

Book E Architecture and Motorola Implementation Standards

The Book E architecture memory and cache definitions are flexible to support a wide variety of embedded implementations. To provide that flexibility, Book E defines many features in a very general way, leaving specific details up to the implementation. To ensure consistency among its Book E cores and devices, Motorola has defined more specific implementation standards. However, these standards still leave many details up to individual implementations. To provide context for those features, this chapter describes aspects of the memory hierarchy and the memory management model defined by Book E and also describes the Motorola implementation standards.

NOTE

Some of the features covered in this chapter, in particular the registers, are described in a very general way that does not include some details that are important to the e500 programmer. There are also slight differences in how some features are defined. For e500-specific details, see Chapter 12, “L1 Caches,” and Chapter 13, “Memory Management Units.”

11.1 Memory and Cache Coherency

The primary objective of a coherent memory system is to provide the same image of memory to all devices using the system. Coherency allows synchronization and cooperative use of shared resources. Otherwise, multiple copies of data corresponding to a memory location, some containing outdated values, could exist in a system, resulting in errors when the outdated values are used. Each memory-sharing device must follow rules for managing the state of its cache. This section describes the coherency mechanisms of the Book E architecture and the cache coherency protocols that the e500 core complex caches support.

Unless specifically noted, the discussion of coherency in this section applies to the core complex data cache only. The instruction cache is not snooped for general coherency with other caches; however, it is snooped when the Instruction Cache Block Invalidate (**icbi**)

instruction is executed by this processor or any processor in the system. See Section 12.3.2, “Instruction Cache Coherency Model,” for more information.

11.1.1 Memory/Cache Access Attributes (WIMGE Bits)

Some memory characteristics can be set on a page basis by using the WIMGE bits in the translation lookaside buffer (TLB) entries. These bits allow both uniprocessor and multiprocessor system designs to exploit numerous system-level performance optimizations. The WIMGE attributes control the following functionalities:

- Write-through (W bit)
- Caching-inhibited (I bit)
- Memory-coherency-required (M bit)
- Guarded (G bit)
- Endianess (E bit)

In addition to the WIMGE bits, the Book E MMU model defines the following attributes on a page basis:

- User-definable (U0, U1, U2, U3)

The WIMGE attributes are programmed by the operating system for each page. The W and I attributes control how the processor performing an access uses its own cache. The M attribute ensures that coherency is maintained for all copies of the addressed memory location. The G attribute prevents speculative loading from the addressed memory location. (An operation is said to be performed speculatively if, at the time that it is performed, it is not known to be required by the sequential execution model.) The E attribute defines the order in which the bytes that comprise a multiple-byte data object are stored in memory (big or little endian).

The WIMGE attributes occupy five bits in the TLB entries for page address translation. The operating system writes the WIMGE bits for each page into the TLB entries in system memory as it maps translations. For more information on TLB entries, see Section 11.3.2, “TLB Entries.”

All combinations of these attributes are supported except combinations that simultaneously specify a region as write-through and caching-inhibited. The write-through and caching-inhibited attributes are mutually exclusive because the write-through attribute permits the data to be in the data cache while the caching-inhibited attribute does not.

Memory that is write-through or caching-inhibited is not intended to be used for general-purpose programming. For example, the **lwarx**, **ldarx**, **stwcx.**, and **stdcx.** instructions may cause the system DSI exception handler to be invoked if they specify a location in memory having either of these attributes. The e500 takes a data storage interrupt if the location is write-through but does not take the interrupt if the location is cache-inhibited.

11.1.1.1 Write-Through Attribute

A store to a write-through ($W = 1$) memory location is performed in main memory and may cause additional locations in memory to be accessed. If a copy of the block containing the specified location is retained in the data cache, the store is also performed in the data cache. A store to write-through memory cannot cause the block to be put in a modified state in the data cache.

When a page in memory is not designated as write-through ($W = 0$), it is considered to be write-back. If some store instructions executed by a given processor access locations in a block as write-through and other store instructions executed by the same processor access locations in that block as write-back, software must ensure that the block cannot be accessed by another processor or mechanism in the system. Also, if a store instruction that accesses a location in the block as write-through is executed when the block is already considered to be modified in the data cache, the block may continue to be considered to be modified in the data cache even if the store causes all modified locations in the block to be written to main memory.

In some processors (not including the e500), accesses caused by separate store instructions that specify locations in write-through memory may be combined into one access. This is called store-gathering. Such combining does not occur if the store instructions are separated by an **msync** or an **mbar**.

11.1.1.2 Caching-Inhibited Attribute

A load instruction that specifies a location in caching-inhibited ($I = 1$) memory is performed to main memory and may cause additional locations in main memory to be accessed unless the specified location is also guarded. An instruction fetch from caching-inhibited memory may cause additional words in main memory to be accessed. No copy of the accessed locations is placed into the caches.

In some processors (not including the e500), non-overlapping accesses caused by separate load instructions that specify locations in caching-inhibited memory may be combined into one access, as may non-overlapping accesses caused by separate store instructions to caching-inhibited memory (that is, store-gathering). Such combining does not occur if the load or store instructions are separated by an **msync** instruction, or by an **mbar** instruction if the memory is also guarded.

11.1.1.3 Memory-Coherence-Required Attribute

Memory coherence refers to the ordering of stores to a single location. Atomic stores to a given location are coherent if they are serialized in some order, and no processor or mechanism is able to observe any subset of those stores as occurring in a conflicting order. This serialization order is an abstract sequence of values; the physical location need not assume each of the values written to it. For example, a processor may update a location

several times before the value is written to physical memory. The result of a store operation is not available to every processor or mechanism at the same instant, and it may be that a processor or mechanism observes only some of the values that are written to a location.

However, when a location is accessed atomically and coherently by all processors and mechanisms, the sequence of values loaded from the location by any processor or mechanism during any interval of time forms a subsequence of the sequence of values that the location logically held during that interval. That is, a processor or mechanism can never load a newer value first and then, later, load an older value.

Memory coherence is managed in blocks called coherence blocks. Their size is implementation-dependent but is usually larger than a word and often the size of a cache block.

When memory coherence is not required ($M = 0$), the hardware need not enforce data coherence for memory accesses initiated by the processor. When memory coherence is required ($M = 1$), the hardware must enforce data coherence for memory accesses initiated by the processor.

Hardware support for the memory-coherence-required attribute is optional for implementations that do not support multiprocessing.

11.1.1.4 Guarded Attribute

When the guarded bit is set, the page is designated as guarded. This setting can be used to protect certain memory areas from read accesses made by the processor that are not dictated directly by the program. If areas of physical memory are not fully populated (in other words, there are holes in the physical memory map within this area), this setting can protect the system from undesired accesses caused by speculative (referred to as ‘out-of-order’ in the architecture specification) load operations that could lead to the generation of the machine check exception. Also, the guarded bit can be used to prevent speculative load operations from occurring to certain peripheral devices that produce undesired results when accessed in this way.

11.1.1.4.1 Definition of Speculative and Out-of-Order Memory Accesses

In the architecture definition, the term ‘out-of-order’ replaced the term ‘speculative’ with respect to memory accesses to avoid a conflict between the word’s meaning in the context of execution of instructions past unresolved branches. The architecture’s use of out-of-order in this context could in turn be confused with the notion of loads and stores being reordered in a weakly ordered memory system.

In the context of memory accesses, this document uses the terms ‘speculative’ and ‘out-of-order’ as follows:

- Speculative memory access. An access to memory that occurs before it is known to be required by the sequential execution model.
- Out-of-order memory access. A memory access performed ahead of one that may have preceded it in the sequential model, such as is allowed by a weakly ordered memory model.

11.1.1.4.2 Performing Operations Speculatively

An operation is said to be nonspeculative if it is guaranteed to be required by the sequential execution model. Any other operation is said to be performed speculatively, which the architecture specification refers to as out of order.

Operations are performed speculatively by hardware on the expectation that the results will be needed by an instruction that will be required by the sequential execution model. Whether the results are needed depends on whether control flow is diverted away from the instruction by an event such as an exception, branch, trap, system call, return from interrupt instruction, or anything else that changes the context in which the instruction is executed.

Typically, the hardware performs operations speculatively when it has resources that would otherwise be idle, so the operation incurs little or no cost. If subsequent events such as branches or exceptions indicate that the operation would not have been performed in the sequential execution model, the processor abandons any results of the operation except as described below.

Most operations can be performed speculatively, as long as the machine appears to follow the sequential execution model. Certain speculative operations are restricted, as follows.

- Stores—A store instruction may not be executed speculatively in a manner such that the alteration of the target location can be observed by other processors or mechanisms.
- Accessing guarded memory—The restrictions for this case are given in Section 11.1.1.4.4, “Speculative Accesses to Guarded Memory.”

No error of any kind other than a machine check exception may be reported due to an operation that is performed speculatively, until such time as it is known that the operation is required by the sequential execution model. The only other permitted side effect (other than machine check) of performing an operation speculatively is that nonguarded memory locations that could be fetched into a cache by nonspeculative execution may be fetched speculatively into that cache.

11.1.1.4.3 Guarded Memory

Memory is said to be well behaved if the corresponding physical memory exists and is not defective, and if the effects of a single access to it are indistinguishable from the effects of multiple identical accesses to it. Data and instructions can be fetched speculatively from well-behaved memory without causing undesired side effects.

Memory is said to be guarded if the G bit is set for the page.

In general, memory that is not well-behaved should be guarded. Because such memory may represent an I/O device or may include non-existent locations, a speculative access to such memory may cause an I/O device to perform incorrect operations or may cause a machine check.

Note that if separate store instructions access memory that is both caching-inhibited and guarded, the accesses are performed in the order specified by the program. If an aligned, load or store that is not a string or multiple access to caching-inhibited, guarded memory has accessed main memory and an external, decrementer, or imprecise-mode floating-point enabled exception is pending, the load or store is completed before the exception is taken.

11.1.1.4.4 Speculative Accesses to Guarded Memory

Accesses for load instructions from guarded memory may be performed speculatively if a copy of the target location is in a cache; in this case, the location may be accessed from the cache or from main memory.

Note that software should ensure that only well-behaved memory is loaded into a cache, either by marking as caching-inhibited (and guarded) all memory that may not be well-behaved, or by marking such memory caching-allowed (and guarded) and referring only to cache blocks that are well-behaved.

11.1.1.4.5 Instruction Accesses: Guarded Memory and No-Execute Memory

The G bit is ignored for instruction fetches, and instructions are speculatively fetched from guarded pages. To prevent speculative instruction fetches from pages that do not contain instructions and are not well-behaved, the page should be designated as no-execute (with the UX/SX page permission bits cleared). If the effective address of the current instruction is mapped to no-execute memory, an ISI exception is generated.

11.1.1.5 Endianness

Objects may be loaded from or stored to memory in byte, half word, word, or double word units. For a particular data length, the load and store operations are symmetrical; a store followed by a load of the same data object yields an unchanged value. Book E makes no guarantees about the order in which the bytes that comprise multiple-byte data objects are stored into memory. The endianness (E) page attribute distinguishes between memory that is big- or little-endian, as described in the following subsections.

Except for instruction fetches, accesses to the same memory location using two effective addresses for which the E bit differs are always permitted. Instruction pages must be flushed from any caches before the E bit can be changed for those addresses. See Section 3.3.3.4, “Byte Ordering,” for more information on big- and little-endian byte ordering.

11.1.1.5.1 Big-Endian Pages

If a stored multiple-byte object is probed by reading its component bytes one at a time using load-byte instructions, the store order may be perceived. If such probing shows that the lowest memory address contains the highest-order byte of the multiple-byte scalar, the next higher sequential address the next least significant byte, and so on, then the multiple-byte object is stored in big-endian form.

Note that strings are not multiple-byte scalars but are interpreted as a series of single-byte scalars. Bytes in a string are loaded from memory using a load string word instruction, starting at the lowest-numbered address, and placed into the target register or registers starting at the left-most byte of the least-significant word. Bytes in a string are stored using a store string word instruction from the source register, starting at the left-most byte of the least-significant word, and placed into memory, starting at the lowest numbered address.

11.1.1.5.2 Little-Endian Pages

Alternatively, if the probing shows that the lowest memory address contains the lowest-order byte of the multiple-byte scalar, the next higher sequential address the next most significant byte, and so on, the multiple-byte object is stored in little-endian form.

11.1.1.5.3 Structure Mapping Examples

The following C programming example defines the data structure *S* used in this section to demonstrate how the bytes that comprise each element (a, b, c, d, e, and f) are mapped into memory. The structure contains scalars (shown in hexadecimal in the comments) and a sequence of characters, shown in single quotation marks.

```
struct {
    int      a;          /* 0x1112_1314          word          */
    double   b;          /* 0x2122_2324_2526_2728 double word */
    char *   c;          /* 0x3132_3334          word          */
    char     d[7];       /* 'L','M','N','O','P','Q','R' array of bytes */
    short    e;          /* 0x5152              half word    */
    int      f;          /* 0x6162_6364          word          */
} S;
```

Figure 11-1 shows the big-endian mapping of the structure. Note that the MSB of each scalar is at the lowest address. The mapping uses padding (indicated by (x)) to align the scalars—4 bytes between elements a and b, 1 byte between d and e, and 2 bytes between e and f. Note that the padding is determined by the compiler, not the architecture.

Contents	11	12	13	14	(x)	(x)	(x)	(x)
Address	00	01	02	03	04	05	06	07

Contents	21	22	23	24	25	26	27	28
Address	08	09	0A	0B	0C	0D	0E	0F

Contents	31	32	33	34	'L'	'M'	'N'	'O'
Address	10	11	12	13	14	15	16	17

Contents	'P'	'Q'	'R'	(x)	51	52	(x)	(x)
Address	18	19	1A	1B	1C	1D	1E	1F

Contents	61	62	63	64	(x)	(x)	(x)	(x)
Address	20	21	22	23	24	25	26	27

Figure 11-1. Big-Endian Mapping of Structure S

Figure 11-2 shows the structure using little-endian mapping, showing double words laid out with addresses increasing from right to left.

Contents	(x)	(x)	(x)	(x)	11	12	13	14
Address	07	06	05	04	03	02	01	00

Contents	21	22	23	24	25	26	27	28
Address	0F	0E	0D	0C	0B	0A	09	08

Contents	'O'	'N'	'M'	'L'	31	32	33	34
Address	17	16	15	14	13	12	11	10

Contents	(x)	(x)	51	52	(x)	'R'	'Q'	'P'
Address	1F	1E	1D	1C	1B	1A	19	18

Contents	(x)	(x)	(x)	(x)	61	62	63	64
Address	27	26	25	24	23	22	21	20

Figure 11-2. Little-Endian Mapping of Structure S —Alternate View

11.1.1.6 Mismatched Memory Cache Attributes

Accesses to the same memory location using two effective addresses for which the write-through required attribute (W bit) differs meet the memory coherence requirements described in Section 11.1.1.1, “Write-Through Attribute,” if the accesses are performed by a single processor. If the accesses are performed by two or more processors, coherence is enforced by the hardware only if the write-through attribute is the same for all the accesses.

Loads, stores, **dcbz** instructions, and instruction fetches to the same memory location using two effective addresses for which the caching-inhibited attribute (I bit) differs must meet the requirement that a copy of the target location of an access to caching-inhibited memory not be in the cache. Violation of this requirement is considered a programming error; software must ensure that the location has not previously been brought into the cache or, if it has, that it has been flushed from the cache. If the programming error occurs, the result of the access is boundedly undefined. It is not considered a programming error if the target location of any other cache management instruction to caching-inhibited memory is in the cache.

Accesses to the same memory location using two effective addresses for which the memory coherence attribute (M bit) differs may require explicit software synchronization before accessing the location with $M = 1$ if the location has previously been accessed with $M = 0$. Any such requirement is system-dependent. For example, in some systems that use bus snooping, no software synchronization may be required. In some directory-based systems, software may be required to execute **dcbf** instructions on each processor to flush all cache entries accessed with $M = 0$ before accessing those locations with $M = 1$.

Accesses to the same memory location using two effective addresses for which the guarded attribute (G bit) differs are always permitted.

Except for instruction fetches, accesses to the same memory location using two effective addresses for which the endian storage attribute (E bit) differs are always permitted as described in Section 11.1.1.5, “Endianness.” Instruction memory locations must be flushed before the endian attribute can be changed for those addresses.

The requirements on mismatched user-defined memory attributes (U0–U3) is implementation-dependent.

11.1.1.7 Coherency Paradoxes and WIMGE

Care must be taken with respect to the use of the WIMGE bits if coherent memory support is desired. Careless programming of these bits may create situations that present coherency paradoxes to the processor. These coherency paradoxes can occur within a single processor or across several processors. It is important to note that, in the presence of a paradox, the operating system software is responsible for correctness.

In particular, a coherency paradox can occur when the state of these bits is changed without appropriate precautions (such as flushing the pages that correspond to the changed bits from the caches of all processors in the system) or when the address translations of aliased real addresses specify different values for certain WIMGE bit values. For more information, see Section 11.1.1.6, “Mismatched Memory Cache Attributes.”

Support for $M = 1$ memory is optional. Cache attribute settings where both $W = 1$ and $I = 1$ are not supported. For all supported combinations of the W, I and M bits, both G and E may be 0 or 1.

The default setting of the WIMGE bits is 0b01000.

11.1.2 Cache Control Instructions

The Book E PowerPC architecture defines instructions for controlling both the instruction and data caches (when they exist).

- Data Cache Block Touch (**dcbt**)
- Data Cache Block Touch for Store (**dcbtst**)
- Data Cache Block Zero (**dcbz**)
- Data Cache Block Store (**dcbst**)
- Data Cache Block Flush (**dcbf**)
- Data Cache Block Allocate (**dcba**)
- Data Cache Block Invalidate (**dcbi**)
- Instruction Cache Block Invalidate (**icbi**)
- Instruction Synchronize (**isync**)
- Instruction Cache Block Touch (**icbt**)

These instructions are described in Section 3.4.1.9.1, “User-Level Cache Instructions,” and Section 3.4.2.2.1, “Supervisor-Level Cache Instruction.” Implementation specific details are described in Section 12.4.1, “Cache Control Instructions.”

Motorola Book E defines the cache block locking APU, which adds the following instructions:

- Data Cache Block Lock Clear (**dcblic**)
- Data Cache Block Touch and Lock Set (**dcbtlic**)
- Data Cache Block Touch for Store and Lock Set (**dcbtstlic**)
- Instruction Cache Block Lock Clear (**icblic**)
- Instruction Cache Block Touch and Lock Set (**icbtlic**)

These instructions are described in Section 10.5.1.1, “Cache Block Lock and Unlock APU Instruction Model.”

11.1.2.1 Self-Modifying Code

When instructions are modified by processors or by other mechanisms, software must ensure that the instruction cache is made consistent with data memory and that the modifications are made visible to the instruction fetching mechanism. The following instruction sequence can be used to accomplish this when the instructions being modified are in memory that is memory-coherence required and one program both modifies the

instructions and executes them. (Additional synchronization is needed when one program modifies instructions that another program will execute.)

The following sequence of instructions synchronizes the instruction stream (using either **dcbst** or **dcbf**):

dcbst (or dcbf)	update memory
msync	wait for update
icbi	remove (invalidate) copy from instruction cache
msync	ensure that ICBI invalidation at icache has completed
isync	remove copy in own instruction buffer

11.1.3 Shared Memory

The architecture supports sharing memory between programs, between different instances of the same program, and between processors and other mechanisms. It also supports access to a memory location by one or more programs using different effective addresses. In these cases memory is shared in blocks that are an integral number of pages. When one physical memory location has different effective addresses, the addresses are said to be aliases. Each application can be granted separate access privileges to aliased pages.

Section 11.1.3.2, “Lock Acquisition and Import Barriers,” gives examples of how **msync** and **mbar** are used to control memory access ordering when memory is shared among programs.

11.1.3.1 Memory Access Ordering

The memory model in Book E for memory access ordering is weakly consistent. This provides an opportunity for improved performance over a model with stronger consistency rules but places the responsibility on the program to ensure that ordering or synchronization instructions are properly placed for the correct execution of the program.

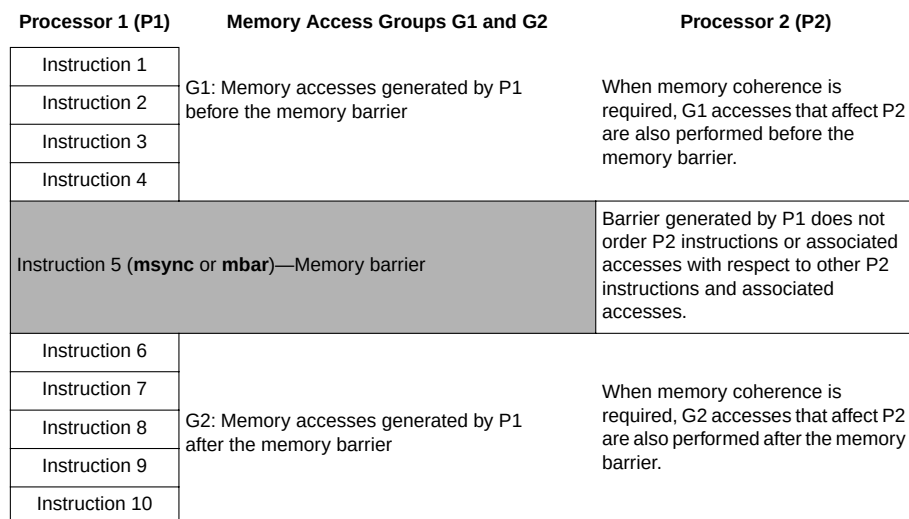
The order in which a processor accesses memory, the order in which those accesses are performed with respect to other processors or mechanisms, and the order in which they are performed in main memory may all be different. Table 11-1 describes how the architecture defines requirements for ordering of loads and stores.

Table 11-1. Load and Store Ordering

Type of Access	Architecture Definition
Load ordering with respect to other loads	The architecture guarantees that loads that are both caching-inhibited (I = 1) and guarded (G = 1) are not reordered with respect to one another. If a load instruction depends on the value returned by a preceding load (because the value is used to compute the effective address specified by the second load), the corresponding memory accesses are performed in program order with respect to any processor or mechanism to the extent required by the associated memory coherence required attributes (that is, the memory coherence required attribute, if any, associated with each access). This applies even if the dependency does not affect program logic (for example, the value returned by the first load is ANDed with zero and then added to the effective address specified by the second load).
Store ordering with respect to other stores	If two store instructions specify memory locations that are both caching inhibited and guarded, the corresponding memory accesses are performed in program order with respect to any processor or mechanism. Otherwise, stores are weakly ordered with respect to one another.
Store ordering with respect to loads	The architecture specifies that an msync or mbar must be used to ensure sequential ordering of loads with respect to stores. Section 12.3.5.3, "Enforcing Store Ordering with Respect to Loads," describes how this is implemented in the e500.

When a processor (P1) executes **msync** or **mbar**, a memory barrier is created that separates applicable memory accesses into two groups, G1 and G2. G1 includes all applicable memory accesses associated with instructions preceding the barrier-creating instruction, and G2 includes all applicable memory accesses associated with instructions following the barrier-creating instruction.

Figure 11-3 shows an example using a two-processor system.

**Figure 11-3. Memory Barrier when Coherency is Required (M = 1)**

The memory barrier ensures that all memory accesses in G1 are performed with respect to any processor or mechanism, to the extent required by the associated memory coherence required attributes (that is, the memory-coherence required attribute, if any, associated with each access), before any memory accesses in G2 are performed with respect to that processor or mechanism.

The ordering enforced by a memory barrier is said to be cumulative if it also orders memory accesses that are performed by processors and mechanisms other than P1, as follows:

- G1 includes all applicable memory accesses by any such processor or mechanism that have been performed with respect to P1 before the memory barrier is created.
- G2 includes all applicable memory accesses by any such processor or mechanism that are performed after a load instruction executed by that processor or mechanism has returned the value stored by a store that is in G2.

Figure 11-4 shows an example of a cumulative memory barrier in a two-processor system.

Processor 1 (P1)	Memory Access Groups G1 and G2	Processor 2 (P2)
P1 Instruction 1	G1: Memory accesses generated by P1 and P2 that affect P1. Includes accesses generated by executing P2 instructions L–O (assuming that the access generated by instruction O occurs before P1's msync is executed).	P2 Instruction L
P1 Instruction 2		P2 Instruction M
P1 Instruction 3		P2 Instruction N
P1 Instruction 4		P2 Instruction O
P1 Instruction 5 (msync)—Cumulative memory barrier applies to all accesses except those associated with fetching instructions following msync .		P2 Instruction P
		P2 Instruction Q
		P2 Instruction R
P1 Instruction 6	G2: Memory accesses generated by P1 and P2. Includes accesses generated by P2 instructions P–X (assuming that the access generated by instruction P occurs after P1's msync is executed) performed after a load instruction executed by P2 has returned the value stored by a store that is in G2.	P2 Instruction S
P1 Instruction 7		P2 Instruction T
P1 Instruction 8		P2 Instruction U
P1 Instruction 9		P2 Instruction V
P1 Instruction 10	The msync memory barrier does not affect accesses associated with instruction fetching that occur after the msync .	P2 Instruction W
P1 Instruction 11		P2 Instruction X

Figure 11-4. Cumulative Memory Barrier

A memory barrier created by **msync** is cumulative and applies to all accesses except those associated with fetching instructions following the **msync**. See the definition of **mbar** in Section 3.4.1.7, “Memory Synchronization Instructions,” for a description of the corresponding properties of the memory barrier created by that instruction.

11.1.3.1.1 Programming Considerations

Because stores cannot be performed out of program order, as described in Book E, if a store instruction depends on the value returned by a preceding load (because the value the load

returns is needed to compute either the effective address specified by the store or the value to be stored), the corresponding accesses are guaranteed to be performed in program order. The same applies whether or not the store instruction executes is dependent upon a conditional branch that in turn depends on the value returned by a preceding load. For example, if a conditional branch depends on a preceding load and that branch chooses between a path that includes a store instruction if the condition is met, that dependent store is not performed unless and until the condition determined by the load is met.

Because instructions following an **isync** cannot execute until all instructions preceding **isync** have completed, if an **isync** follows a conditional branch instruction that depends on the value returned by a preceding load instruction, that load is performed before any loads caused by instructions following the **isync**. This is true even if the effects of the dependency are independent of the value loaded (for example, the value is compared to itself and the branch tests $CRn[EQ]$), and even if the branch target is the next sequential instruction.

Except for the cases described above and earlier in this section, data and control dependencies do not order memory accesses. Examples include the following:

- If a load specifies the same memory location as a preceding store and the location is not caching inhibited, the load may be satisfied from a store queue (a buffer into which the processor places stored values before presenting them to the memory subsystem) and not be visible to other processors and mechanisms. As a result, if a subsequent store depends on the value returned by the load, the two stores need not be performed in program order with respect to other processors and mechanisms.
- Because a store conditional instruction may complete before its store is performed, a conditional branch instruction that depends on the $CR0$ value set by a store conditional instruction does not order that store with respect to memory accesses caused by instructions that follow the branch.

For example, in the following sequence, the **stw** instruction is the **bc** instruction's target:

```
stwcx.  
bc  
stw
```

For the **stwcx.** to complete, it must update the architected $CR0$ value, even though its store may not have been performed. The architecture does not require that the store generated by the **stwcx.** must be performed before the store generated by the **stw**.

- Because processors may predict branch target addresses and branch condition resolution, control dependencies (branches, for example) do not order memory accesses except as described above. For example, when a subroutine returns to its caller, the return address may be predicted, with the result that loads caused by

instructions at or after the return address may be performed before the load that obtains the return address is performed.

Some processors implement nonarchitected duplicates of architected resources such as GPRs, CR fields, and the LR, so resource dependencies (for example, specification of the same target register for two load instructions) do not force ordering of memory accesses.

Examples of correct uses of dependencies, **msync**, and **mbar** to order memory accesses can be found in Section B.1, “Synchronization.”

Because the memory model is weakly consistent, the sequential execution model as applied to instructions that cause memory accesses guarantees only that those accesses appear to be performed in program order with respect to the processor executing the instructions. For example, an instruction may complete, and subsequent instructions may be executed, before memory accesses caused by the first instruction have been performed. However, for a sequence of atomic accesses to the same memory location for which memory coherence is required, the definition of coherence guarantees that the accesses are performed in program order with respect to any processor or mechanism that accesses the location coherently, and similarly if the location is one for which caching is inhibited.

Because accesses to caching-inhibited memory are performed in main memory, memory barriers and dependencies on load instructions order such accesses with respect to any processor or mechanism even if the memory is not designated as requiring memory coherence.

11.1.3.1.2 Programming Examples

Example 1 shows cumulative ordering of memory accesses preceding a memory barrier, and the second example illustrates cumulative ordering of memory accesses following a memory barrier. Assume that locations X, Y, and Z initially contain the value 0.

Example 1:

- Processor A stores the value 1 to location X.
- Processor B loads from location X obtaining the value 1, executes an **msync**, then stores the value 2 to location Y.
- Processor C loads from location Y obtaining the value 2, executes an **msync**, then loads from location X.

Example 2:

- Processor A stores the value 1 to location X, executes an **msync**, then stores the value 2 to location Y.
- Processor B loops, loading from location Y until the value 2 is obtained, then stores the value 3 to location Z.

- Processor C loads from location Z obtaining the value 3, executes an **msync**, then loads from location X. In both cases, cumulative ordering dictates that the value loaded from location X by processor C is 1.

11.1.3.2 Lock Acquisition and Import Barriers

An import barrier is an instruction or instruction sequence that prevents memory accesses caused by instructions following the barrier from being performed before memory accesses that acquire a lock have been performed. An import barrier can be used to ensure that a shared data structure protected by a lock is not accessed until the lock has been acquired. An **msync** instruction can always be used as an import barrier, but the approaches shown below generally yield better performance because they order only the relevant memory accesses.

11.1.3.2.1 Acquire Lock and Import Shared Memory

If **lwarx** and **stwcx.** are used to obtain the lock, an import barrier can be constructed by placing an **isync** immediately following the loop containing the **lwarx** and **stwcx.**. The following example uses the Compare and Swap primitive (see Section B.1.1, “Synchronization Primitives”) to acquire the lock.

In this example, it is assumed that the address of the lock is in GPR 3, the value indicating that the lock is free is in GPR 4, the value to which the lock should be set is in GPR 5, the old value of the lock is returned in GPR 6, and the address of the shared data structure is in GPR 9.

```
loop:  lwarx    r6,0,r3          # load lock and reserve
      cmpw    r4,r6            # skip ahead if
      bne-    wait             # lock not free
      stwcx.  r5,0,r3          # try to set lock
      bne-    loop             # loop if lost reservation
      isync                   # import barrier
      lwz     r7,data1(r9)      # load shared data
      .
      .
wait:  ...                      #wait for lock to free
```

The second **bne-** does not complete until CR0 has been set by the **stwcx.**. The **stwcx.** does not set CR0 until it has completed (successfully or unsuccessfully). The lock is acquired when the **stwcx.** completes successfully. Together, the second **bne-** and the subsequent **isync** create an import barrier that prevents the load from data1 from being performed until the branch is resolved to be not taken.

11.1.3.2.2 Obtain Pointer and Import Shared Memory

If **lwarx** and **stwcx.** are used to obtain a pointer into a shared data structure, an import barrier is not needed if all the accesses to the shared data structure depend on the value

obtained for the pointer. The following example uses the Fetch and Add primitive (see Section B.1.1, “Synchronization Primitives”) to obtain and increment the pointer.

In this example, it is assumed that the address of the pointer is in GPR 3, the value to be added to the pointer is in GPR 4, and the old value of the pointer is returned in GPR 5.

```
loop:  lwarx    r5,0,r3          # load pointer and reserve
       add     r0,r4,r5        # increment the pointer
       stwcx.  r0,0,r3        # try to store new value
       bne-    loop           # loop if lost reservation
       lwz     r7,data1(r5)    # load shared data
```

The load from data1 cannot be performed until the **lwarx** loads the pointer value into GPR 5. The load from data1 may be performed out-of-order before the **stwcx.** But if the **stwcx.** fails, the branch is taken and the value returned by the load from data1 is discarded. If the **stwcx.** succeeds, the value returned by the load from data1 is valid even if the load is performed out-of-order, because the load uses the pointer value returned by the instance of the **lwarx** that created the reservation used by the successful **stwcx.**

An **isync** could be placed between the **bne-** and the subsequent **lwz**, but no **isync** is needed if all accesses to the shared data structure depend on the value returned by the **lwarx**.

11.1.3.3 Atomic Memory References

The Book E architecture defines the Load Word and Reserve Indexed (**lwarx**) and the Store Word Conditional Indexed (**stwcx.**) instructions to provide an atomic update function for a single, aligned word of memory. These instructions can be used to develop a rich set of multiprocessor synchronization primitives. Note that atomic memory references constructed using **lwarx/stwcx.** instructions depend on the presence of a coherent memory system for correct operation. These instructions should not be expected to provide atomic access to noncoherent memory.

The **lwarx** instruction performs a load word from memory operation and creates a reservation for the same reservation granule that contains the accessed word. The reservation granularity is implementation-dependent. Section 12.3.5.4, “Atomic Memory References,” describes how the e500 implements this feature.

The **lwarx** instruction makes a non-specific reservation with respect to the executing processor and a specific reservation with respect to other masters. This means that any subsequent **stwcx.** executed by the same processor, regardless of address, cancels the reservation. Also, any bus write or invalidate operation from another processor to an address that matches the reservation address cancels the reservation.

11.2 Book E MMU Architecture

This section describes the memory management structure defined by Book E. The memory management approach defined by the Book E architecture is suited for desktop applications

and has the simplicity and flexibility necessary for embedded applications. Book E supports demand-paged virtual memory as well as a variety of other management schemes that depend on precise control of effective-to-real address translation and flexible memory protection. Address translation misses and protection faults cause precise exceptions. Sufficient information is available to correct the fault and restart the faulting instruction.

Each program on a 64-bit or 32-bit implementation can access 2^{64} or 2^{32} bytes of effective address (EA) space, subject to limitations imposed by the operating system. In a typical Book E system, each program's EA space is a subset of a larger virtual address (VA) space managed by the operating system.

Each effective (logical) address is translated to a real (physical) address before being used to access physical memory or an I/O device. The hardware accomplishes this using the address translation mechanism described in Section 11.2.3, "Address Translation," Section 11.3.1, "Address Translation and the Motorola Standard," and Section 13.2, "Effective-to-Real Address Translation." The operating system manages the physically addressed resources of the system by setting up the tables used by the address translation mechanism.

The Book E architecture divides the effective address space into pages. The page represents the granularity of effective address translation, permission control, and memory/cache attributes. Up to 16 page sizes (1 Kbyte, 4 Kbyte, 16 Kbyte, 64 Kbyte, 256 Kbyte, 1 Mbyte, 4 Mbyte, 16 Mbyte, 64 Mbyte, 256 Mbyte, 1 Gbyte, 4 Gbyte, 16 Gbyte, 64 Gbyte, 256 Gbyte, 1 Tbyte) may be simultaneously supported. For an effective-to-real address translation to exist, a valid entry for the page containing the effective address must be in a translation lookaside buffer (TLB). Addresses for which no TLB entry exists cause TLB miss exceptions (instruction or data TLB error interrupts).

The instruction addresses generated by a program and the addresses used by load, store, and cache management instructions are effective addresses. However, in general, the physical memory space may not be large enough to map all the virtual pages used by the currently active applications. With support provided by hardware, the operating system can attempt to use the available real pages to map a sufficient set of virtual pages for an application. If a sufficient set is maintained, paging activity is minimized, therefore maximizing performance.

The operating system can restrict access to virtual pages by selectively granting permissions for user state read, write, and execute, and supervisor state read, write, and execute on a per-page basis. These permissions can be set up for a particular system (for example, program code might be execute-only, data structures may be mapped as read/write/no-execute) and can also be changed by the operating system based on application requests and operating system policies.

11.2.1 Virtual Address

Book E defines a virtual address space composed of the effective address of an access, the 1-bit current address space (AS) of the access and the 32-bit process ID (PID) of an access, as shown in Figure 11-5. The following subsections describe the selection of AS and PID for an effective address, both used to construct the virtual address for an access.

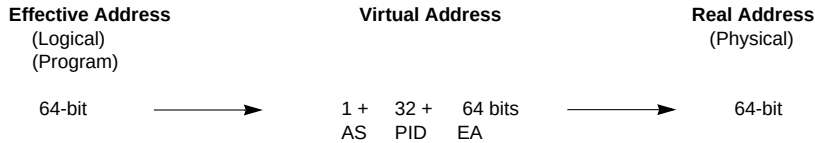


Figure 11-5. Virtual Address Space in Book E

11.2.1.1 Address Spaces

Instruction accesses are generated by sequential instruction fetches or due to a change in program flow (branches and interrupts). Data accesses are generated by load, store, and cache management instructions.

The Book E architecture defines two address spaces for instruction accesses and two address spaces for data accesses. The current address space for instruction or data accesses is determined by the value of MSR[IS] and MSR[DS], respectively, as shown in Figure 11-6. The address space indicator (the value of either MSR[IS] or MSR[DS], as appropriate) is used in addition to the effective address generated by the processor for translation into a physical address by the TLB mechanism.

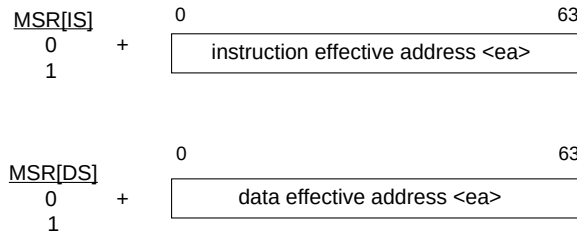


Figure 11-6. Current Address Space

Because MSR[IS] and MSR[DS] are cleared when an interrupt occurs, an address space value of zero can be used to denote interrupt-related address spaces, or possibly all system software address spaces; an address space value of one can be used to denote non interrupt-related address spaces, or possibly all user address spaces.

11.2.1.1.1 Instruction Address Spaces

The two effective instruction address spaces are defined by the value of MSR[IS], and instruction fetch addresses are translated from the effective address space specified by the current value of MSR[IS]. Changing the value of MSR[IS] is considered a context-altering operation, requiring a context synchronization operation to follow it. When a context synchronizing event occurs, any prefetched instructions are discarded and instructions are re-fetched using the then-current state of MSR[IS] and the then-current program counter. See Section 3.3.3.5.2, “Context Synchronization,” for more information on the definition of context synchronizing events.

Instructions are not fetched from memory designated by the TLB mechanism as no-execute (UX = 0 or SX = 0, see Section 11.2.4.1, “Execute Access Permission”). If the effective address of the current instruction is mapped to no-execute memory, an instruction storage interrupt (ISI) is generated.

Note that mapping a page as no-execute does not affect instruction caches in the system (or any instructions resident in unified caches). Thus, if an instruction is loaded into a cache when its effective address is mapped to execute permitted memory, and the execute permissions for that page are later changed to no-execute, any instructions fetched before the no-execute mapping remain in the cache until explicitly evicted by an **icbi** instruction or through the cache’s replacement policy. However, attempted execution of such instructions still results in an ISI. Thus, for example, the operating system can change the designation of an application’s instruction pages to no-execute without having to first flush instruction cache blocks that map to these pages.

11.2.1.1.2 Data Address Spaces

The two effective data address spaces are defined by the value of MSR[DS] and data is accessed to/from the effective address space specified by the current value of MSR[DS]. As is the case with MSR[IS], changing the value of MSR[DS] is considered a context-altering operation, requiring a context synchronization operation to follow it. When a context synchronizing event occurs, subsequent accesses are made using the new state of MSR[DS] (see Section 3.3.3.5.2, “Context Synchronization”).

Data can be read from a page, provided the user read (UR) permission bit is set in the TLB for a user access, or the supervisor read (SR) bit is set for a supervisor access. Likewise, data write access permissions are determined by the user write (UW) and supervisor write (SW) permission bits. If permissions are violated as defined in Section 11.2.4.2, “Read Access Permission,” or Section 11.2.4.3, “Write Access Permission,” the appropriate interrupt is taken.

11.2.1.2 Process ID

As described in ____ reg mod ch ____, the Book E architecture defines that a PID value be associated with each effective address (instruction or data) generated by the processor. At the Book E level, one 32-bit PID register maintains the value of the PID for the current process. This PID value is used to construct a virtual address for accessing memory.

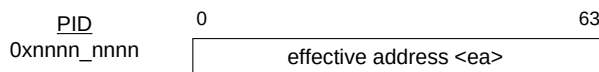


Figure 11-7. Current PID Value

11.2.2 Book E TLB Concept

A TLB is defined by Book E to be the hardware resource that contains address translations, implements protections, and maintains the memory/cache attributes for pages. The organization of a TLB (for example, unified versus separate instruction and data, TLB hierarchies, TLB associativity, number of entries) is implementation-dependent.

Book E does not define a page table format nor a TLB format. Instead, it defines certain attributes associated with a TLB entry. It is up to the implementation to define the exact form and function of a TLB entry. The architected attributes include the following:

- Address space
- Process ID
- Effective page number
- Real page number
- Permission bits
- WIMGE bits
- Page size representation

Book E MMUs may be hardware or software managed. Because the architecture is targeted for embedded applications, most implementations are likely to be software managed. This allows many different operating systems to be ported quickly because memory management data structures may be in any form, and requiring specific porting only at points where hardware intervention is required (such as a TLB entry replacement operation). Porting of operating system software is thus less complex and requires less memory.

The Book E architecture does not specify exactly how TLB entries are read or written except that TLB read, write, search, and invalidate instructions are defined as follows:

- TLB Write Entry—**tlbwe**
- TLB Read Entry—**tlbre**

- TLB Search—**tlbsx** rA,rB
- TLB Invalidate All—**tlbivax** rA,rB

The definitions of these instructions dictate that an implementation-dependent method be used to read or write TLB information. This allows many possible implementations.

In addition, the Book E architecture defines the following instruction that provides ordering for the effects of all **tlbivax** instructions:

- TLB Synchronize—**tlbsync**

The TLB entry fields defined by the Book E architecture are described in Table 11-2.

Table 11-2. Book E Defined TLB Fields

Field	Description
V	Valid 0 This TLB entry is not valid for translation. 1 This TLB entry is valid for translation.
TID	Translation ID. Identifies the process ID for this TLB entry. Translation IDs are compared with process IDs during translation to help select a TLB entry. This field is used to share pages among processes. If TID = 0, this entry is globally shared. Thus, if a TLB entry's TID field is zero, it globally matches all PID values.
TS	Translation space. Identifies which address space is valid for this TLB entry. TS is compared with the current value of MSR[IS] for instruction accesses and MSR[DS] for data accesses. This allows for an efficient change of address space when a transition from user to supervisor mode occurs.
SIZE	Page size. This field describes the size of the page. Not all Motorola implementations will support varying page sizes. Some TLB arrays support a range of page sizes while other TLB arrays only support a fixed size.
EPN	Effective page number. Describes the effective (or logical) starting address of the page. The number of bits that are valid (used in translation) depend on the size of the page. EPN is compared with the effective address being translated to partially select a TLB entry.
WIMGE	Memory and cache access attributes. Describe the characteristics of memory accesses to the page and the subsequent treatment of that data with respect to the memory subsystem (caches and bus transactions). The WIMGE bits function as they were defined in the original PowerPC architecture. The E bit is new in the Book E architecture and allows the selection of little-endian operation on a page-by-page basis. For further explanation of the WIMGE attributes, see Section 11.1.1, "Memory/Cache Access Attributes (WIMGE Bits)." W Write-through required (1 = write-through required) I Caching inhibited (1 = page is caching inhibited) M Memory coherence required (1 = page is memory-coherence required) G Guarded (1 = page is guarded) E Endianness (0 = page is big endian, 1 = page is little endian)
RPN	Real page number. Describes the physical starting address of the page. The real page number replaces the effective page address resulting in a translated address. The number of bits used in translation depend on the size of the page (that is, the same number of bits as the EPN for a given page translation).
UX	User execute permission 0 Instructions may not be executed out of this page when the processor is in user mode (that is, MSR[PR] = 1). When a user program attempts to execute instructions out of this page, an ISI interrupt occurs. 1 Instructions may be executed out of this page when the processor is in user mode (that is, MSR[PR] = 1).

Table 11-2. Book E Defined TLB Fields (continued)

Field	Description
SX	Supervisor execute permission 0 Instructions may not be executed out of this page when the processor is in supervisor mode (that is, MSR[PR] = 0). When a supervisor program attempts to execute instructions out of this page, an ISI interrupt occurs. 1 Instructions may be executed out of this page when the processor is in supervisor mode (i.e. MSR[PR] = 0).
UW	User write permission 0 Stores may not be performed to this page when the processor is in user mode (that is, MSR[PR] = 1). If a user program attempts to store instructions to this page, a DSI interrupt occurs. 1 Stores may be performed to this page when the processor is in user mode (that is, MSR[PR] = 1).
SW	Supervisor write permission 0 Stores may not be performed to this page when the processor is in supervisor mode. If a supervisor program attempts to store to this page, a DSI interrupt occurs. 1 Stores may be performed to this page when the processor is in supervisor mode.
UR	User read permission 0 Loads may not be performed from this page when the processor is in user mode. If a user program attempts to load from this page, a DSI interrupt occurs. Note that in user mode, instructions are allowed to be fetched even when UR = 0. 1 Loads may be performed from this page when the processor is in user mode.
SR	Supervisor read permission 0 Loads may not be performed from this page when the processor is in supervisor mode. If a supervisor program attempts to load from this page, a DSI interrupt occurs. Note that in supervisor mode, instructions are allowed to be fetched even when SR = 0. 1 Loads may be performed from this page when the processor is in supervisor mode.
U0–U3	User page attribute bits. These bits are implementation-dependent. In general, Motorola implementations leave these as extra memory bits associated with a TLB entry to be used by system software.

11.2.3 Address Translation

The effective address is the untranslated address either for an instruction fetch address or for a data address that is calculated as a result of a load, store, or cache management instruction. The effective address, concatenated with the address space value of the MSR[IS] or MSR[DS], is compared to the appropriate number of bits of the EPN field (depending on the page size) and the TS field of the TLB entry. If a match occurs, that TLB entry is a candidate for a translation match. In addition to a match in the EPN field and TS, a matching TLB entry must match with the current process ID of the access.

Figure 11-8 shows the translation match logic for the effective address plus its attributes (collectively called the virtual address) and how it is compared with the corresponding fields in the TLB entries.

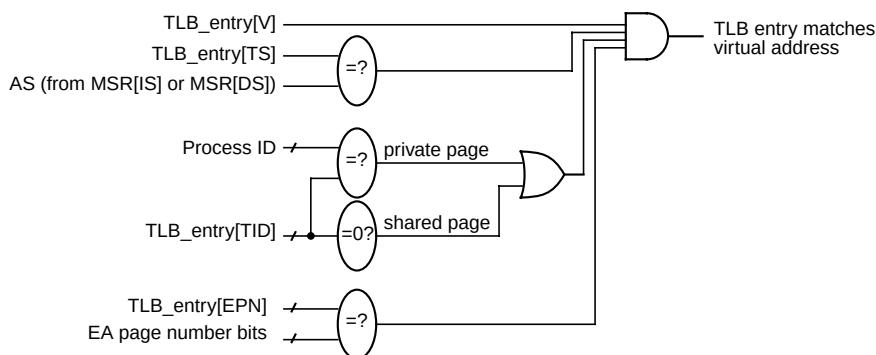


Figure 11-8. Virtual Address and TLB-Entry Comparison

On a TLB hit, the generation of the physical address for a 64-bit implementation occurs as shown in Figure 11-9.

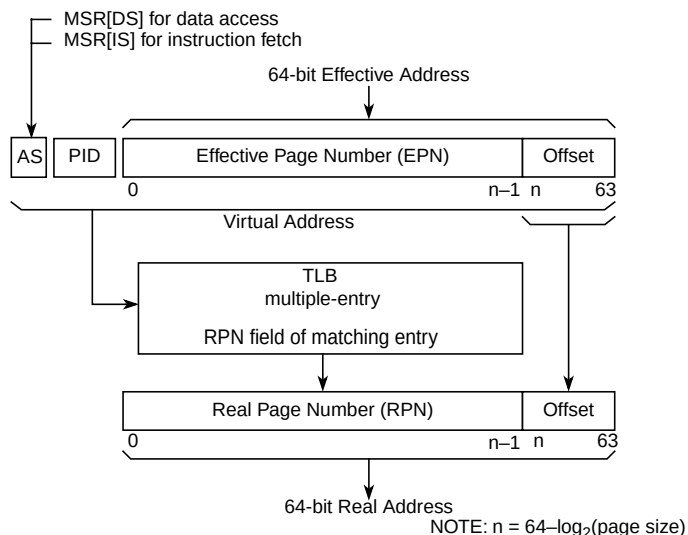


Figure 11-9. Effective-to-Real Address Translation—64-Bit Implementations

The generation of the physical address for a 32-bit implementation occurs as shown in Figure 11-10.

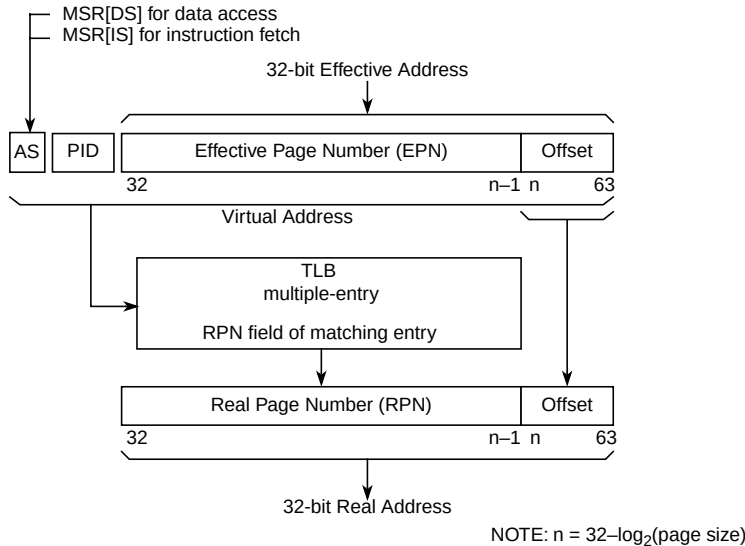


Figure 11-10. Effective-to-Real Address Translation—32-Bit Implementations

11.2.3.1 Page Size and Effective Address Bits Compared

The page size defined for a TLB entry determines how many bits of the effective address are compared with the corresponding EPN field in the TLB entry as shown in Table 11-3.

Table 11-3. Page Size and EPN Field Comparison

SIZE Field	Page Size (4^{SIZE} Kbytes)	64-Bit Implementations EA to EPN Comparison (Bits 0–53; $2 \times \text{SIZE}$)	32-Bit Implementations EA to EPN Comparison (Bits 32–53; $2 \times \text{SIZE}$)
0b0000	1 Kbyte	EA[0–53] = ? EPN[0–53]	EA[32–53] = ? EPN[32–53]
0b0001	4 Kbyte	EA[0–51] = ? EPN[0–51]	EA[32–51] = ? EPN[32–51]
0b0010	16 Kbyte	EA[0–49] = ? EPN[0–49]	EA[32–49] = ? EPN[0–49]
0b0011	64 Kbyte	EA[0–47] = ? EPN[0–47]	EA[32–47] = ? EPN[32–47]
0b0100	256 Kbyte	EA[0–45] = ? EPN[0–45]	EA[32–45] = ? EPN[32–45]
0b0101	1 Mbyte	EA[0–43] = ? EPN[0–43]	EA[32–43] = ? EPN[32–43]
0b0110	4 Mbyte	EA[0–41] = ? EPN[0–41]	EA[32–41] = ? EPN[32–41]
0b0111	16 Mbyte	EA[0–39] = ? EPN[0–39]	EA[32–39] = ? EPN[32–39]
0b1000	64 Mbyte	EA[0–37] = ? EPN[0–37]	EA[32–37] = ? EPN[32–37]
0b1001	256 Mbyte	EA[0–35] = ? EPN[0–35]	EA[32–35] = ? EPN[32–35]
0b1010	1 Gbyte	EA[0–33] = ? EPN[0–33]	EA[32–33] = ? EPN[32–33]
0b1011	4 Gbyte	EA[0–31] = ? EPN[0–31]	—

Table 11-3. Page Size and EPN Field Comparison (continued)

SIZE Field	Page Size (4^{SIZE} Kbytes)	64-Bit Implementations EA to EPN Comparison (Bits 0–53; $2 \times \text{SIZE}$)	32-Bit Implementations EA to EPN Comparison (Bits 32–53; $2 \times \text{SIZE}$)
0b1100	16 Gbyte	EA[0–29] = ? EPN[0–29]	—
0b1101	64 Gbyte	EA[0–27] = ? EPN[0–27]	—
0b1110	256 Gbyte	EA[0–25] = ? EPN[0–25]	—
0b1111	1 Tbyte	EA[0–23] = ? EPN[0–23]	—

11.2.3.2 Permission Attribute Comparison

As part of the translation process, the selected TLB entry provides the access permission bits (UX, SX, UW, SW, UR, SR), and memory/cache attributes (U0, U1, U2, U3, W, I, M, G, and E) for the access. These bits specify whether or not the access is allowed and how the access is to be performed.

If a matching TLB entry has been identified, Book E provides an access permission mechanism that selectively grants shared access, grants execute access, grants read access, grants write access, and prohibits access to areas of memory based on a number of criteria. Book E defines the permission bits in TLB entries as follows:

- SR—Supervisor read permission.
- SW—Supervisor write permission.
- SX—Supervisor execute permission.
- UR—User read permission.
- UW—User write permission.
- UX—User execute permission.

If the virtual address translation comparison with TLB entries was successful, the permission bits for the matching entry are checked as shown in Figure 11-11. If the access is not allowed by the access permission mechanism, the processor generates an instruction or data storage interrupt (ISI or DSI).

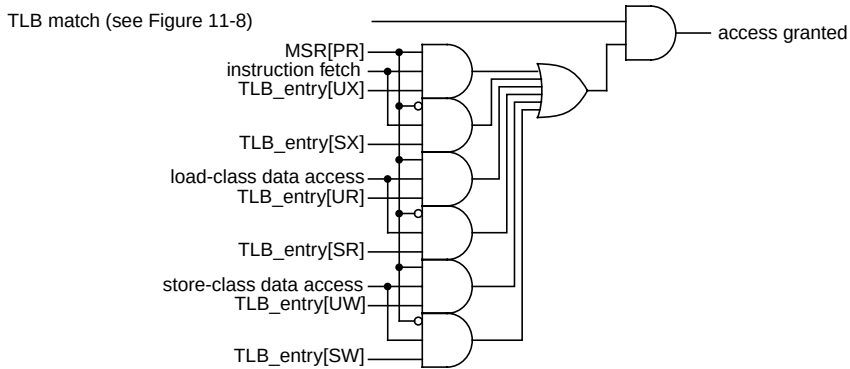


Figure 11-11. Granting of Access Permission

The permission attributes defined by Book E are defined in detail in Section 11.2.4, “Permission Attributes.”

11.2.3.3 Page Size and Real Address Generation

If no virtual address match occurs, the translation fails and a TLB miss exception occurs. Depending on the access type (instruction or data address), either the instruction TLB error interrupt or the data TLB error interrupt is taken.

Otherwise, the real page number (RPN) field of the matching TLB entry provides the translation for the effective address of the access. Based on the setting of the SIZE field of the matching TLB entry, the RPN field replaces the corresponding most-significant n bits of the effective address where $n = 64 - \log_2(\text{page size})$ for 64-bit implementations and $n = 32 - \log_2(\text{page size})$ for 32-bit implementations. Note that the untranslated bits must be zero in the RPN field.

Table 11-4. Real Address Generation

SIZE Field	Page Size (4^{SIZE} Kbytes)	64-Bit Implementations		32-Bit Implementations	
		RPN Bits Required to be Equal to 0	Real Address	RPN Bits Required to be Equal to 0	Real Address
0b0000	1 Kbyte	none	RPN[0–53] EA[54–63]	none	RPN[32–53] EA[54–63]
0b0001	4 Kbyte	RPN[52–53] = 0	RPN[0–51] EA[52–63]	RPN[52–53] = 0	RPN[32–51] EA[52–63]
0b0010	16 Kbyte	RPN[50–53] = 0	RPN[0–49] EA[50–63]	RPN[50–53] = 0	RPN[32–49] EA[50–63]
0b0011	64 Kbyte	RPN[48–53] = 0	RPN[0–47] EA[48–63]	RPN[48–53] = 0	RPN[32–47] EA[48–63]
0b0100	256 Kbyte	RPN[46–53] = 0	RPN[0–45] EA[46–63]	RPN[46–53] = 0	RPN[32–45] EA[46–63]
0b0101	1 Mbyte	RPN[44–53] = 0	RPN[0–43] EA[44–63]	RPN[44–53] = 0	RPN[32–43] EA[44–63]
0b0110	4 Mbyte	RPN[42–53] = 0	RPN[0–41] EA[42–63]	RPN[42–53] = 0	RPN[32–41] EA[42–63]

Table 11-4. Real Address Generation (continued)

SIZE Field	Page Size (4 ^{SIZE} Kbytes)	64-Bit Implementations		32-Bit Implementations	
		RPN Bits Required to be Equal to 0	Real Address	RPN Bits Required to be Equal to 0	Real Address
0b0111	16 Mbyte	RPN[40–53] = 0	RPN[0–39] EA[40–63]	RPN[40–53] = 0	RPN[32–39] EA[40–63]
0b1000	64 Mbyte	RPN[38–53] = 0	RPN[0–37] EA[38–63]	RPN[38–53] = 0	RPN[32–37] EA[38–63]
0b1001	256 Mbyte	RPN[36–53] = 0	RPN[0–35] EA[36–63]	RPN[36–53] = 0	RPN[32–35] EA[36–63]
0b1010	1 Gbyte	RPN[34–53] = 0	RPN[0–33] EA[34–63]	RPN[34–53] = 0	RPN[32–33] EA[34–63]
0b1011	4 Gbyte	RPN[32–53] = 0	RPN[0–31] EA[32–63]	—	
0b1100	16 Gbyte	RPN[30–53] = 0	RPN[0–29] EA[30–63]	—	
0b1101	64 Gbyte	RPN[28–53] = 0	RPN[0–27] EA[28–63]	—	
0b1110	256 Gbyte	RPN[26–53] = 0	RPN[0–25] EA[26–63]	—	
0b1111	1 Tbyte	RPN[24–53] = 0	RPN[0–23] EA[24–63]	—	

11.2.4 Permission Attributes

The permission attributes defined in Book E are shown in Table 11-5 and described in the following subsections.

Table 11-5. Permission Control for Instruction, Data Read, and Data Write Accesses

Access Type	MSR[PR]	TLB[UX]		TLB[SX]		TLB[UR]		TLB[SR]		TLB[UW]		TLB[SW]	
		0	1	0	1	0	1	0	1	0	1	0	1
Instruction Fetch	0	—	—	ISI	√	—	—	—	—	—	—	—	—
	1	ISI	√	—	—	—	—	—	—	—	—	—	—
Data Read (load)	0	—	—	—	—	—	—	DSI	√	—	—	—	—
	1	—	—	—	—	DSI	√	—	—	—	—	—	—
Data Write (store)	0	—	—	—	—	—	—	—	—	—	—	DSI	√
	1	—	—	—	—	—	—	—	—	DSI	√	—	—

11.2.4.1 Execute Access Permission

The UX and SX bits of the TLB entry control execute access to the corresponding page.

Instructions may be fetched and executed from a page in memory while the processor is operating in user mode (MSR[PR] = 1) if the UX access control bit for that page is set. If the UX access control bit is cleared, instructions from that page are not fetched and they are not placed into any cache while the processor is in user mode.

Instructions may be fetched and executed from a page in memory while in supervisor mode ($\text{MSR}[\text{PR}] = 0$) if the SX access control bit for that page is set. If the SX access control bit is cleared, instructions from that page are not fetched and are not placed into any cache while in the processor is in supervisor mode.

If the sequential execution model calls for the execution of an instruction from a page that is not enabled for execution (i.e. $\text{UX} = 0$ when $\text{MSR}[\text{PR}] = 1$ or $\text{SX} = 0$ when $\text{MSR}[\text{PR}] = 0$), an execute access control exception type instruction storage interrupt (ISI) is taken.

11.2.4.2 Read Access Permission

The UR and SR bits of the TLB entry control read access to the corresponding page.

Load operations (including load-class cache management instructions) are permitted from a page in memory while the processor is in user mode ($\text{MSR}[\text{PR}] = 1$) if the UR access control bit for that page is set. If the UR access control bit is cleared, execution of the load instruction is suppressed and a read access control exception type data storage interrupt (DSI) is taken.

Similarly, load operations (including load-class cache management instructions) are permitted from a page in memory while in supervisor state ($\text{MSR}[\text{PR}] = 0$) if the SR access control bit for that page is set. If the SR access control bit is cleared, execution of the load instruction is suppressed and a read access control exception type data storage interrupt (DSI) is taken.

11.2.4.3 Write Access Permission

The UW and SW bits of the TLB entry control write access to the corresponding page.

Store operations (including store-class cache management instructions) are permitted to a page in memory while the processor is in user mode ($\text{MSR}[\text{PR}] = 1$) if the UW access control bit for that page is set. If the UW access control bit is cleared, execution of the store instruction is suppressed and a write access control exception type data storage interrupt (DSI) is taken.

Similarly, store operations (including store-class cache management instructions) are permitted to a page in memory while the processor is in supervisor mode ($\text{MSR}[\text{PR}] = 0$) if the SW access control bit for that page is set. If the SW access control bit is cleared, execution of the store instruction is suppressed and a write access control exception type data storage interrupt (DSI) is taken.

11.2.4.4 Permission Control and Cache Management Instructions

The **dcbi** and **dcbz** instructions are treated as stores because they can change data (or cause loss of data by invalidating a modified line). As such, they both can cause write access control exception type DSIs.

The **dcba** instruction is treated as a store because it can also change data. As such, it can also cause a write access control exception. However, these exceptions do not result in a data storage interrupt and if a permission violation occurs, the instruction execution completes, but the allocate operation is merely cancelled (essentially, a no-op).

The **icbi** instruction is treated as a load with respect to permissions checking. As such, it can cause a read access control exception type data storage interrupt.

The **dcbt**, **dcbtst**, and **icbt** instructions are treated as loads with respect to permissions checking. As such, they can cause read access control exceptions. However, such exceptions do not result in data storage interrupts and if a permission violation occurs, the instruction execution completes, but the operation is cancelled (essentially, a no-op).

The **dcbf** and **dcbst** instructions are treated as loads with respect to permissions checking. Flushing or storing a line from the cache is not considered a store because the store has already been performed to update the cache and the **dcbf** or **dcbst** instruction is only updating the copy in main memory. Like load instructions, these instructions can cause read access control exception type data storage interrupts.

Table 11-6 summarizes the exceptions cases caused by the cache management instructions due to permissions violations.

Table 11-6. Permission Control and Cache Instructions

Instruction	Can Cause Read Permission Violation Exception?	Can Cause Write Permission Violation Exception?
dcba	No	Yes ¹
dcbf	Yes	No
dcbi	No	Yes
dcbst	Yes	No
dcbt	Yes ¹	No
dcbtst	Yes ¹	No
dcbz	No	Yes
icbi	Yes	No
icbt	Yes ¹	No

¹ **dcbt**, **dcbtst**, and **icbt** may cause a read access control exception but do not result in a data storage interrupt (DSI).

Throughout this chapter, references to load instructions include the cache management and other instructions that are stated in the instruction descriptions to be treated as a load, and references to store instructions include the cache management and other instructions that are treated as a store.

11.2.4.5 Permissions Control and String Instructions

When the string length is zero, neither **lswx** nor **stswx** can cause data storage interrupts due to permissions violations.

11.2.4.6 Use of Permissions to Maintain Page History

The Book E architecture TLB entry definition does not define bits for maintaining page history information. The U0–U3 bits in the TLB entries can be used by software for storing history information, but the e500 core complex always ignores these bits internally.

Page changed bit status can be implemented in the system software by disabling write permissions to all pages. The first attempt to write to the page results in a data storage interrupt. At this point, system software can record the page changed bit in memory, update the TLB entry permission to allow writes to that page, and return to the user program allowing further writes to the page to proceed without exception.

11.2.5 Crossing Page Boundaries

Care must be taken with single instruction accesses (load/stores) that cross page boundaries. Examples are **lmw/stmw** instructions and misaligned accesses on implementations that support misaligned load/stores. Architecturally, each of the parts of the access that cross the natural boundary of the access size (half word, word, double word) are treated separately with respect to exception conditions. Additionally, these types of instructions may optionally partially complete. For example, a store word instruction that crosses a page boundary because it is misaligned to the last half word of a page might actually store the first 16 bits because the access was permitted, but produce a DSI or data TLB error exception because the second 16 bits in the next page were not valid or they were protected. An implementation may choose to suppress the first 16-bit store or perform it.

11.3 Motorola Book E MMU Architecture

Because much of the Book E architecture leaves implementation details, and thus a specific programming model, up to the implementation, Motorola has defined further standards for Book E processors. The Motorola standard for Book E MMUs establishes a common way of implementing Book E processors to provide a programming model that is consistent across all products in the family.

This reduces the software efforts required in porting to a new processor because the programming model is less likely to change. Thus, the standard defines configuration information for features such as TLBs, caches, and other entities that have standard forms, but differing attributes (like cache sizes and associativity) such that a single software implementation can be created that works efficiently for all implementations of a class.

The Motorola Book E MMU standard defines functions and structures that are visible to the execution model of the processor. These consist of the following definitions:

- The TLB, from a programming point of view, consists of zero or more TLB arrays, each of which may have differing characteristics.
- The logical-to-physical address translation mechanism.
- Methods and effects of changing and manipulating TLB arrays.
- Configuration information available to the operating system that describes the structure and form of the TLB arrays and translation mechanism.

To assist or accelerate translation, implementations may contain other TLB structures not visible to the programming model. These structures and the methods for using them are not explicitly defined in the architecture or the Motorola standard, but they may be considered at the operating system level because they may affect an implementation's performance. See Chapter 13, "Memory Management Units," for more detailed information about the TLBs on the e500 core complex.

11.3.1 Address Translation and the Motorola Standard

Translating an effective address to a real address is defined by Book E to require four elements:

- The PID value. The Motorola Book E standard defines up to 15 PID registers and defines the PID value as 12 bits long. The contents of PID0–PID14 are used in the compare with TID values in the TLB entries. (The Book E architecture refers to only one PID register.)
- The address space value. Depending on the type of translation (instruction or data), MSR[IS] or MSR[DS] is used.
- The TLB entries in the TLB arrays
- The effective address being translated

The following subsections describe these elements as they are further defined by the Motorola Book E standard.

11.3.1.1 Variable Number of PID Registers and Sharing

The 15 PID registers supported by the Motorola Book E standard are implemented as SPR registers set by system software, and collectively reflect the process ID of the currently

executing context. The values of all PID registers are compared with the TID field of all candidate TLB entries. If the contents of any PID register matches a TID field in any TLB entry, that TLB entry is a candidate for a possible match for translation. The system maintains multiple PID values in order to allow the sharing of TLB entries for pages that are shared among multiple execution contexts. For example, system software may assign PID0 to contain the unique process ID (for private mappings for the current processes) and may assign PID1 to contain the unique process ID for a common set of shared libraries.

Note that Book E defines the value of all zeros for a TID field in a TLB entry as an entry that is globally shared. Thus, when PID values (up to 12 bits for Motorola devices) are compared to the TID fields in the TLB arrays for matches, if a TLB entry contains all zeros in the TID field, it globally matches all PID values. See Chapter 13, “Memory Management Units,” for more detailed information about the PID registers implemented on the e500 core complex.

11.3.1.2 Match Criteria for TLB Entries

TLB arrays contain TLB entries that are used to match any address presented for translation. All TLB entries for any given implementation are candidates for any given translation. The TLB itself is unordered with respect to the various elements used in address translations, and regardless of implementation, should be considered to perform the translation comparison with all entries in parallel.

There should be only one valid matching translation for a given effective address, PID value, and address space value. If the TLB contains more than one matching entry, it is considered a programming error, and the behavior of any such translation is undefined. The processor will likely go into the checkstop state or take a machine check interrupt in this case.

The following fields are compared in the TLB entries:

- V—The matching entry must have the V bit set.
- TS—The address space identifier used for translation. The appropriate bit of MSR[IS] or MSR[DS] must match the TS bit for a matching entry.
- TID—The contents of a PID register must match the TID field of a matching entry, or the TID field must be all zeros for a matching entry.
- EPN—The appropriate number of bits (depending on the page size) of the effective address being translated is compared to the EPN field of the TLB entry.

If a match occurs on all the fields listed above, the physical address is formed by replacing the effective page number in the effective address with the value in the RPN field of the matching TLB entry. The number of bits in the page number depends on the page size for that TLB entry.

11.3.1.3 Translation Algorithms

The following algorithm describes how translation operates at the Motorola Book E level:

```

ea = effective address
if translation is an instruction address then
    as = MSR[IS]
else    // data address translation
    as = MSR[DS]
for all TLB entries
    if ! TLBV then
        next    // compare next TLB entry
    if as != TLBTS then
        next
    if TLBTID == 0 then
        goto pid_match
    for all PID registers
        if this PID register == TLBTID then
            goto pid_match
    endfor
    next    // no PIDs matched

pid_match:    // translation match
    mask = ~(1024 << (2 * TLBTSIZE)) - 01)
    if (ea & mask) != TLBEPN then
        next    // no address match
    real address = TLBRPN | (ea & ~mask)    // real address computed
end translation -- success
endfor
end translation -- tlbmiss

```

The algorithm for the granting of permission is as follows:

```

if MSRPR == 0 then
    x = TLBSX
    r = TLBSR
    w = TLBSW
else
    x = TLBUX
    r = TLBUR
    w = TLBUW

if instruction fetch address then
    if x == 0 then
        Instruction Storage Interrupt
    else    // data access
        if data read (load) then
            if r == 0 then
                Data Storage Interrupt
            else    // write access (store)
                if w == 0 then
                    Data Storage Interrupt

```

11.3.2 TLB Entries

The software-visible TLB is subdivided into zero or more TLB arrays. Each array must contain TLB entries that share the same characteristics. Each TLB array contains one or more TLB entries. Each entry has specific fields that correspond to fields in the seven MMU assist (MAS) registers, described in Section 11.3.5, “MMU Assist (MAS) Registers.” Some TLB fields are architected in Book E and others are architected in the Motorola Book E standard. Note that Book E architected fields may have restrictions or enhancements imposed by the Motorola Book E architecture for the Motorola implementations.

11.3.2.1 IPROT TLB Entry Field

Only the IPROT TLB entry field is architected by the Motorola Book E. The IPROT field designates TLB entries as protected from certain kinds of invalidation. TLB invalidation and the IPROT field are described further in Section 11.3.4, “TLB Entry Invalidation.”

11.3.3 Reading and Writing TLB Entries

All TLB entries are updated by executing **tlbwe** instructions. At the time of **tlbwe** execution, the MMU assist registers (MAS0–MAS6), a set of SPRs defined by the Motorola Book E standard, are used to index a specific TLB entry. The MAS registers also contain the information that is written to the indexed entry, such that they serve as the ports into the TLBs, as shown in Figure 11-12. The contents of the MAS registers are described in Section 11.3.5, “MMU Assist (MAS) Registers.”

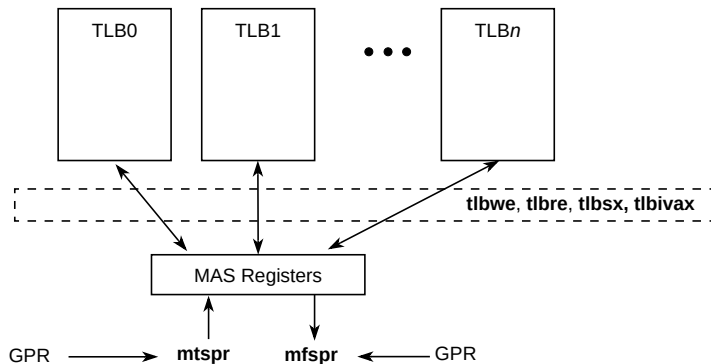


Figure 11-12. TLBs Accessed Through MAS Registers and TLB Instructions

Similarly, TLB entries are read by executing **tlbre** instructions. At the time of **tlbre** execution, the MAS registers are used to index a specific TLB entry and upon completion of the **tlbre** instruction, the MAS registers contain the contents of the indexed TLB entry. To read or write TLB entries, the MAS registers are first loaded by system software using **mtspr** instructions and then the desired **tlbre** or **tlbwe** instructions must be executed.

Note that $RA = 0$ is a preferred form for **tlbsx** and that some Motorola implementations, such as the e500, take an illegal instruction exception program interrupt if $RA \neq 0$.

11.3.3.1 Reading TLB Entries

For TLB read operations, $MAS0[TLBSEL]$ identifies the TLB array to be accessed. In addition, depending on the value of $MAS0[TLBSEL]$, the $MAS0[ESEL]$ and $MAS2[EPN]$ fields must be set to identify the particular TLB entry in the array that is to be accessed. After execution of the **tlbre** instruction, the MAS registers contain the contents of the TLB entry specified. System software may examine or store the contents of the TLB entry by executing **mfspr** instructions to move information from the MAS registers to GPRs.

11.3.3.2 Writing TLB Entries

For TLB write operations, the TLB entry to be accessed is selected in the same manner as for a **tlbre** instruction by setting the values of the $MAS0[TLBSEL]$, $MAS0[ESEL]$, and $MAS2[EPN]$ fields appropriately. The rest of the MAS register contents must also be written by system software to achieve the desired writing of a TLB entry; when MAS register contents are appropriately set, the execution of a **tlbwe** instruction causes the contents of the MAS registers to be copied into the selected TLB entry. The contents of the MAS registers are not changed as a result of **tlbwe** instruction execution. Default values (in $MAS4$) and search PID values (in $MAS5$ and $MAS6$) are not required to be written for **tlbwe** and do not change the contents of the TLB entry being written.

In general, **tlbwe** should be preceded by an **msync** instruction to ensure that any translated stores that might be in the store queues are completed before a TLB entry is changed (particularly important if the **tlbwe** causes the current page to be re-assigned).

In addition, **tlbwe** should be followed by an **isync** instruction to ensure that the newly written contents of the TLB are used to translate instructions that follow the **tlbwe** instruction. This is especially important if translations being changed affect the current instruction or data stream. For example, if no **isync** is performed and the current instruction translation is changed, an undefined number of instructions may have been fetched using the translation that existed before the **tlbwe** instruction. If system software can ensure that no changed translation is used until after a context synchronizing event (for example, **isync** or **rfi**), an **isync** is not required to immediately follow a **tlbwe**. However, it is considered good practice to ensure that the change has been successfully made and using the **isync** immediately is strongly recommended.

11.3.4 TLB Entry Invalidations

Book E also defines a TLB invalidate instruction, **tlbivax**, that can be used for TLB entry invalidation in this processor and other processors that are kept coherent with this processor. TLB entries can be invalidated in one of the following ways:

- As the result of a **tlbwe** instruction that clears the valid bit in the entry.
- As a result of a **tlbivax** instruction or from a received broadcast invalidation resulting from a **tlbivax** on another processor in an SMP system.
- As a result of a flash invalidate operation initiated by writing to bits in MMUCSR0 (see Section 2.12.2, “MMU Control and Status Register 0 (MMUCSR0)”).

In both multiprocessor and uniprocessor systems, invalidations initiated by the execution of a **tlbivax** instruction can occur on a wider set of TLB entries than specifically intended. That is, depending on the implementation, an effective address presented for invalidation may cause invalidation of TLB entries in addition to the targeted TLB entry. This is because some attributes of the TLB entry to be invalidated may not be fully specified to the hardware at invalidate time. So, in the case of a set-associative TLB array, it may be that all indexed entries (all ways) in the TLB array for a particular effective address are invalidated as a result of a **tlbivax**. In SMP systems, the invalidation address is also broadcast globally to all processors in the system as a result of the **tlbivax**, and the hardware implementation may impose other caveats causing additional invalidations to occur in this case. The architecture assures that the intended TLB is invalidated, but does not guarantee it will be the only one. This phenomenon is referred to as generous invalidates.

A TLB entry invalidated by explicitly clearing the valid bit of a TLB entry with a **tlbwe** instruction is guaranteed to invalidate only the addressed TLB entry. Invalidate operations occurring from **tlbivax** instructions or from the multiprocessor broadcasts as a result of **tlbivax** instructions may cause generous invalidates.

An implementation may provide a method to protect against generous invalidations. This is important because certain effective memory regions must be properly mapped at all times. For example, the instruction memory region that serves as the exception handler for MMU faults must always be mapped. If this region has no valid mapping, an MMU exception can not be handled because the first address of the exception handler causes another MMU exception.

To prevent this from occurring, the Motorola Book E standard specifies the invalidate protection (IPROT) bit for TLB entries. If IPROT is set in a given entry, that entry is protected from invalidations resulting from **tlbivax** instructions, as a result of a broadcast invalidation from another processor in an SMP system, or as a result of a flash invalidate operation initiated through writing to MMUCSR0. TLB entries with the IPROT bit set can only be invalidated by explicitly writing the TLB entry and specifying a 0 for the valid bit.

Operating systems need to use great care when using protected TLB entries, particularly in SMP systems. An SMP system that contains protected (IPROT set) TLB entries on other processors requires a cross-processor interrupt or some other synchronization mechanism to assure that each processor performs the required invalidation by clearing the valid bits of its own TLB entries (using **tlbwe**) in a synchronized way with respect to the other processors in the system.

Note that not all TLB arrays in a given implementation will implement the IPROT attribute. It is likely that implementations that are suitable for demand-page environments will implement it for only a single array, while not implementing it for other on-chip TLB arrays. See Section 13.3.2.1, “IPROT Invalidation Protection in TLB1,” for information on how this is implemented on the e500 core complex.

11.3.5 MMU Assist (MAS) Registers

(Motorola Implementation Standard Definition)

The Motorola Book E standard specifies seven MAS registers (MAS0–MAS6) for use by software to manipulate and query the TLB. SPRs 624 through 630 are reserved for this purpose. MAS registers may be either 64 bits or 32 bits wide, depending on whether the given processor is implemented as a 64- or 32-bit processor. As SPRs, MAS registers can only be accessed by system software in supervisor mode.

The MAS registers are used to transfer information to and from TLB entries as a result of system software executing explicit instructions to perform the transfer, or as a result of certain exception conditions. Specifically, transfers to or from the MAS registers and the TLB entries occur as a result of any of the following conditions:

- Execution of a **tlbre** instruction
- Execution of a **tlbsx** instruction
- Occurance of a permissions violation (causing an instruction storage interrupt or data storage interrupt), or a TLB miss exception (which causes either an instruction TLB error interrupt or data TLB error interrupt).

Field assignments in the MAS registers are the same for both 64- and 32-bit implementations except that the larger address space addressed by 64-bit implementations requires the use of larger fields for addressing. This simplifies creating common system software between 64- and 32-bit processors. Because of differences in address sizes, one set of instructions cannot work for both 64- and 32-bit implementations; however, it is possible to either have run-time checks for the use of the instructions, or relegate the differences to simple callable functions.

The MAS registers contain the following three types of information:

- Fields that correspond to the contents of TLB entries
- Specific fields that identify the TLB entry for which a MAS register operation applies
- Default TLB information that specifies certain fields in the MAS registers to be loaded with default information when exception conditions occur.

MAS register formats are shown in the following subsections as they are defined by the Motorola Book E standard. Note that implementations can more specifically define the

MAS register fields; see Section 13.7.1, “e500 MAS Registers,” for more information on the MAS register definitions for the e500 core complex.

11.3.5.1 MAS Register Formats and Bit Definitions

Figure 11-13 describes the format of MAS0 as defined by the Motorola implementation standards. Note that the e500 definition for this register is more specific and is also described in Section 2.12.5.1, “MAS Register 0 (MAS0).”

	32	33	34	35	36		47	48		51	52		63
Field	—		TLBSEL			ESEL			—			NV	
Reset	All zeros												
R/W	R/W												
SPR	SPR 624												

Figure 11-13. MAS Register 0 (MAS0) Format—Motorola Implementation Standard Definition

The MAS0 fields are described in Table 11-7.

Table 11-7. MAS0 Field Descriptions—Motorola Implementation Standard Definition

	Bit	Name	Descriptions
Motorola Implementation Standard	32–33	—	Reserved, should be cleared.
	34–35	TLBSEL ¹	Selects TLB for access 00 TLB0 01 TLB1 10 TLB2; reserved (must not be used) on e500. 11 TLB3; reserved (must not be used) on e500.
	36–47	ESEL ¹	Entry select. Number of entry in selected array to be used for tlbwe . This field is also updated on TLB error exceptions (misses), and tlbsx hit and miss cases. Only certain bits are valid, depending on the array selected in TLBSEL. Other bits should be 0. (ESEL allows definition of an array that can have up to 4,096 selections.)
	48–51	—	Reserved, should be cleared.
	52–63	NV ¹	Next victim. Defined as a 12-bit field by the Motorola standard. Can be used to identify the next victim to be targeted for a TLB miss replacement operation for those TLBs that support the NV field. If the TLB selected by MAS0[TLBSEL] does not support the NV field, then this field is undefined. The specific meaning of this field is implementation-dependent.

¹ Devices may implement a smaller number of bits for this field, with the bits used being right-justified.

Figure 11-14 shows the format of MAS1 as defined by the Motorola implementation standards. Note that the e500 definition for this register is more specific and is described in Section 2.12.5.2, “MAS Register 1 (MAS1).”

	32	33	34	35	36		47	48	50	51	52		55	56		63
Field	V	IPROT	—			TID		—	TS		TSIZE		—			
Reset	All zeros															
R/W	R/W															
SPR	SPR625															

Figure 11-14. MAS Register 1 (MAS1) Format—Motorola Implementation Standard Definition

The MAS1 fields are described in Table 11-8.

Table 11-8. MAS1 Field Descriptions—Motorola Implementation Standard Definition

	Bits	Name	Descriptions
Motorola Implementation Standard	32	V	TLB valid bit 0 This TLB entry is invalid. 1 This TLB entry is valid.
	33	IPROT	Invalidate protect. Set to protect this TLB entry from invalidate operations due to the execution of the tlbivax instruction. Note that not all TLB arrays are necessarily protected from invalidation with IPROT. Arrays that support invalidate protection are denoted as such in the TLB configuration registers. 0 Entry is not protected from invalidation 1 Entry is protected from invalidation as described in Section 11.3.4, “TLB Entry Invalidations.”
	34–35	—	Reserved, should be cleared.
	36–47	TID ¹	Translation identity. 12-bit field that defines the process ID for this TLB entry. TID is compared with the current process IDs of the effective address to be translated. A TID value of 0 defines an entry as global and matches with all process IDs.
	48–50	—	Reserved, should be cleared.
	51	TS	Translation space. This bit is compared with the IS or DS fields of the MSR (depending on the type of access) to determine if this TLB entry may be used for translation.
	52–55	TSIZE	Translation size. This 4-bit field defines the page size of the TLB entry. For TLB arrays that contain fixed-size TLB entries, this field is ignored. For variable page size TLB arrays, the page size is 4 ^{TSIZE} Kbytes. Note that the Motorola Book E standard supports all 16 page sizes.
	56–63	—	Reserved, should be cleared.

¹ Devices may implement a smaller number of bits for this field, with the bits used being right-justified.

Figure 11-15 shows the format of MAS2 as defined by the Motorola implementation standards. Note that the e500 definition for this register is more specific and is described in Section 2.12.5.3, “MAS Register 2 (MAS2).”

	32	51	52	56	57	58	59	60	61	62	63		
Field	EPN				—		X0	X1	W	I	M	G	E
Reset	All zeros												
R/W	R/W												
SPR	SPR626												

Figure 11-15. MAS Register 2 (MAS2) Format—Motorola Implementation Standard Definition

The MAS2 fields are described in Table 11-9.

Table 11-9. MAS2 Field Descriptions—Motorola Implementation Standard Definition

Motorola Implementation Standard	Bits	Name	Description
	32–51	EPN	Effective page number. Depending on page size, only the bits that are associated with a page boundary are valid. Bits that represent offsets within a page are ignored and should be zero.
	52–54	—	Reserved for implementation-specific use.
	55–56	—	Reserved, should be cleared.
	57	X0	Implementation-dependent page attribute
	58	X1	Implementation-dependent page attribute
	59	W	Write-through 0 This page is considered write-back with respect to the caches in the system. 1 All stores performed to this page are written through the caches to main memory.
	60	I	Caching-inhibited 0 Accesses to this page are considered cacheable. 1 The page is considered caching-inhibited. All loads and stores to the page bypass the caches and are performed directly to main memory.
	61	M	Memory coherence required 0 Memory coherence is not required. 1 Memory coherence is required. This allows loads and stores to this page to be coherent with loads and stores from other processors (and devices) in the system, assuming all such devices are participating in the coherence protocol.
	62	G	Guarded 0 Accesses to this page are not guarded and can be performed before it is known if they are required by the sequential execution model. 1 All nonspeculative loads and stores to this page are performed (that is, only loads and stores that are known to be required are performed and all such loads and stores). A read or write to a caching-inhibited page affects only the memory element specified by the operation.
	63	E	Endianness. Determines endianness for the corresponding page. Little-endian operation is true little endian, which differs from the modified little-endian byte ordering model optionally available in previous devices that implement the original PowerPC architecture. See Section 11.1.1.5, “Endianness.” 0 The page is accessed in big-endian byte order. 1 The page is accessed in true little-endian byte order.

Figure 11-16 shows the format of MAS3 as defined by the Motorola implementation standards. Note that the e500 definition for this register is the same as described here.

	32		51	52	53	54		57	58	59	60	61	62	63
Field	RPN					—	U0–U3		UX	SX	UW	SW	UR	SR
Reset	All zeros													
R/W	R/W													
SPR	SPR627													

Figure 11-16. MAS Register 3 (MAS3) Format—Motorola Implementation Standard Definition

The MAS3 fields are described in Table 11-10.

Table 11-10. MAS3 Field Descriptions—Motorola Implementation Standard Definition

	Bits	Name	Description
Motorola Implementation Standard	32–51	RPN	Real page number. Depending on page size, only the bits associated with a page boundary are valid. Bits that represent offsets within a page are ignored and should be zero.
	52–53	—	Reserved, should be cleared.
	54–57	U0–U3	User bits. These bits are associated with a TLB entry and can be used by system software. For example, these bits may be used to hold information useful to a page scanning algorithm or be used to mark more abstract page attributes.
	58–63	PERMIS	Permission bits (UX, SX, UW, SW, UR, SR). User and supervisor read, write, and execute permission bits. See Table 11-2 for more information on the page permission bits as they are defined by Book E.

Figure 11-17 shows the format of MAS4 as defined by the Motorola implementation standards. Note that the e500 definition for this register is more specific and is described in Section 2.12.5.5, “MAS Register 4 (MAS4).”

	32	33	34	35	36		43	44		47	48		51	52		55	56	57	58	59	60	61	62	63
Field	—	TLBSELD		—		TIDSELD		—		TSIZED		—	X0D	X1D	WD	ID	MG	DE	ED					
Reset	All zeros																							
R/W	R/W																							
SPR	SPR628																							

Figure 11-17. MAS Register 4 (MAS4) Format—Motorola Implementation Standard Definition

The MAS4 fields are described in Table 11-11.

Table 11-11. MAS4 Field Descriptions—Motorola Implementation Standard Definition

	Bits	Name	Description
Motorola Implementation Standard	32–33	—	Reserved, should be cleared.
	34–35	TLBSEL ¹	TLBSEL default value. A 2-bit field that specifies the default value to be loaded in MAS0[TLBSEL] on a TLB miss exception. See Section 11.3.6.1, “TLB Miss Exception Types,” for more information.
	36–43	—	Reserved, should be cleared.
	44–47	TIDSEL ¹	TID default selection value. A 4-bit field that specifies which of the current PID registers should be used to load the MAS1[TID] field on a TLB miss exception.
	48–51	—	Reserved, should be cleared.
	52–55	TSIZED	Default TSIZE value. Specifies the default value to be loaded into MAS1[TSIZE] on a TLB miss exception.
	56	—	Reserved, should be cleared.
	57	X0D	Default X0 value. Specifies the default value to be loaded into MAS2[X0] on a TLB miss exception.
	58	X1D	Default X1 value. Specifies the default value to be loaded into MAS2[X1] on a TLB miss exception.
	59	WD	Default W value. Specifies the default value to be loaded into MAS2[W] on a TLB miss exception.
	60	ID	Default I value. Specifies the default value to be loaded into MAS2[I] on a TLB miss exception.
	61	MD	Default M value. Specifies the default value to be loaded into MAS2[M] on a TLB miss exception.
	62	GD	Default G value. Specifies the default value to be loaded into MAS2[G] on a TLB miss exception.
	63	ED	Default E value. Specifies the default value to be loaded into MAS2[E] on a TLB miss exception.

¹ Devices may implement a smaller number of bits for this field, with the bits used being right-justified.

Figure 11-18 describes the format of MAS5 as defined by the Motorola implementation standards. Note that the e500 does not implement MAS5.

	32	35 36	47 48	51 52	63
Field	—	SPID2	—	SPID3	
Reset	All zeros				
R/W	R/W				
SPR	SPR620				

Figure 11-18. MAS Register 5 (MAS5) Format—Motorola Implementation Standard Definition

The MAS5 fields are described in Table 11-12.

Table 11-12. MAS5 Field Descriptions¹—Motorola Implementation Standard Definition

Motorola Implementation Standard	Bits	Name	Description
	32–35	—	Reserved, should be cleared.
	36–47	SPID2 ²	Search PID2. Specifies the value of PID2 to be used when searching the TLB during execution of tlbsx .
	48–51	—	Reserved, should be cleared.
	52–63	SPID3 ²	Search PID3. Specifies the value of PID3 to be used when searching the TLB during execution of tlbsx .

¹ This register is not implemented on the e500

² Devices may implement a smaller number of bits for this field, with the bits used being right-justified.

Figure 11-19 shows the format of MAS6 as defined by the Motorola implementation standards. Note that the e500 definition for this register is more specific and is described in Section 2.12.5.6, “MAS Register 6 (MAS6).”

	32	35	36	39	40	47	48	50	51	62	63
Field	—				SPID0		—			SPID1	SAS
Reset	All zeros										
R/W	R/W										
SPR	SPR 630										

Figure 11-19. MAS Register 6 (MAS6)—Motorola Implementation Standard Definition

The MAS6 fields are described in Table 11-13.

Table 11-13. MAS6 Field Descriptions—Motorola Implementation Standard Definition

Bits	Name	Description
32–35	—	Reserved, should be cleared.
36–47	SPID0 ¹	Search PID0. Specifies the value of PID0 used when searching the TLB during execution of tlbsx .
48–50	—	Reserved, should be cleared.
51–62	SPID1 ¹	Search PID1. Specifies the value of PID1 used when searching the TLB during execution of tlbsx .
63	SAS	Address space (AS) value for searches. Specifies the value of AS used when searching the TLB (during execution of tlbsx).

¹ Devices may implement a smaller number of bits for this field, with the bits used being right-justified.

11.3.6 MAS Registers and Exception Handling

When translation-related exceptions occur, the MAS registers are preloaded by the hardware with information that is likely to be needed by the interrupt handler to handle the

fault. For a TLB miss exception, some MAS register fields are loaded with default information specified in MAS4. System software should set up the default information in MAS4 prior to allowing exceptions. In most cases, system software sets this up once depending on its scheme for handling page faults. This simplifies the handling of translation-related exceptions. The specific fields of the MAS registers and the contents loaded for each exception type are described in the following subsections.

11.3.6.1 TLB Miss Exception Types

The Book E architecture defines that a TLB miss exception is caused when a virtual address for an access does not match with that of any on-chip TLB entry. This condition causes one of the following:

- An instruction TLB error interrupt or
- A data TLB error interrupt.

11.3.6.1.1 Instruction TLB Error Interrupt Settings

An instruction TLB error interrupt occurs when the the virtual address associated with an instruction address (fetch) does not match any valid entry in the TLB (i.e. the address for the instruction cannot be translated). In addition to the values automatically written to the MAS registers (described in Section 11.3.6.1.3, “TLB Miss Exception MAS Register Settings”), SRR0 contains the address of the instruction that caused the instruction TLB error. This SRR0 value is used to identify the effective address for handling the exception as well as the address to return to when system software has resolved the exception condition by writing a new TLB entry.

11.3.6.1.2 Data TLB Error Interrupt Settings

A data TLB error interrupt occurs when the virtual address associated with a data reference from a load, store, or cache management instruction does not match any valid entry in the TLB (i.e. the address of the data item of a load or store instruction cannot be translated). In addition to the values automatically written to the MAS registers (described in Section 11.3.6.1.3, “TLB Miss Exception MAS Register Settings”), the effective address of the data access that caused the exception is automatically loaded in the data exception address register (DEAR). Also, SRR0 contains the address of the instruction that caused the data TLB error and its value is used to identify the address to return to when system software has resolved the exception condition (by writing a new TLB entry).

11.3.6.1.3 TLB Miss Exception MAS Register Settings

When either an instruction or data TLB error interrupt occurs, the TLB information and selection fields of the MAS registers are loaded with default values from other MAS registers to assist in processing the exception. The intention is that the common case of a page fault generally requires only system software to load the RPN (corresponding to the

physical address that will be used for this page), and the access permissions and the defaults can be used for the remaining MAS fields.

The processor may use the next victim (NV) field from the TLB array to select which TLB entry should be used for the new translation. The method used to select the candidate TLB for replacement (the next victim) is implementation-dependent and may vary on different Motorola Book E implementations. In any case, software is free to choose any TLB entry for the replacement (software can overwrite the value in MAS0[ESEL]).

The Motorola Book E standard defines the fields set in the MAS registers at exception time for an instruction or data TLB error interrupt as shown in Table 11-14.

Table 11-14. MAS Settings for an Instruction or Data TLB Error Interrupt

MAS Field	Value
TLBSEL	Set to value in TLBSELD (default). This defines the TLB array to be used for the new TLB entry that will be written.
ESEL	May be set to an implementation-dependent value, usually based on the NV field of the array selected by TLBSELD, if that TLB supports the NV function. If the selected TLB does not support NV, the value loaded into ESEL is undefined.
NV	Set to an implementation-dependent value to select which TLB entry to replace on the next TLB miss. The NV field of the TLB array is updated by the value of NV in the MAS registers when tlbwe is executed.
V	Set to 1
IPROT	Cleared to 0
TID	Set to the contents of the PID register referenced by TIDSELD. That is, if TIDSELD contains the value 1, the contents of PID1 are written to the TID field.
TS	Set to the value of the IS or DS bit in the MSR at the time of the exception (that is, the MSR that described the context that was running when the exception occurred).
TSIZE	Set to TSIZED
EPN	Set to the effective page number of the instruction or data address causing the exception. The number of bits of the page number is implementation-dependent, but should be consistent with the TLB array selected if the TLB has a fixed page size. If the TLB array selected by TLBSELD contains variable-sized pages, the value for EPN is undefined.
WIMGE and X bits	Set to corresponding default values in the MAS registers
RPN	Cleared to 0
Permissions	SR, UR, UW, SW, UX, SX cleared to 0 (no permissions); note that U0–U3 are unchanged.

All other MAS register values are unchanged.

11.3.6.2 Permissions Violation Exception Types

The Book E architecture also defines that a permissions violation exception is caused when an effective address for an access matches with a TLB entry but the permission attributes

in the matching TLB entry do not allow the access to proceed, as described in Section 11.2.4, “Permission Attributes.” This condition causes either:

- An instruction storage interrupt (ISI) or
- A data storage interrupt (DSI).

11.3.6.2.1 Instruction Storage Interrupt Settings

An instruction storage interrupt occurs when the effective address associated with an instruction address (fetch) matches a valid entry in the TLB but one of the permission bits in the TLB does not allow the instruction fetch. In this case, SRR0 contains the address of the instruction that caused the instruction TLB error and this value is used to identify the effective address for handling the exception as well as the address to return to when system software has resolved the exception condition (by writing a new TLB entry).

11.3.6.2.2 Data Storage Interrupt Settings

A data storage interrupt occurs when the effective address associated with a data reference from a load or store instruction matches with a valid entry in the TLB but one of the permission bits in the TLB does not allow the data access. In this case, the effective address of the data access that caused the exception is contained in the data exception address register (DEAR). This address is used by system software to identify the address that caused the exception. Also, SRR0 contains the address of the instruction that caused the data storage interrupt and its value is used to identify the address to return to when system software has resolved the exception condition by writing a new TLB entry.

11.3.6.3 MAS Register Update Summary for Exceptions, tlbsx, and tlbre

Table 11-15 summarizes MAS register fields updates from the perspective of the Motorola Book E as a result of various events. Note that the e500 implementation further defines how certain MAS fields are set on exceptions as described in Section 13.7.2, “MAS Register Updates.”

Table 11-15. MMU Assist Register Field Updates—Motorola Book E Standard

MAS Register Bit/Field	Value Loaded for Each Case						
	Instr/Data TLB Error	tlbsx Hit	tlbsx Miss	ISI	DSI	tlbre	tlbwe
TLBSEL	TLBSELD	Which TLB hit	TLBSELD	—	—	—	—
ESEL	If TLBSELD supports NV: TLB0[NV] else, undefined	Number of entry that hit	If TLBSELD supports NV: TLB0[NV] else, undefined	—	—	—	—

Table 11-15. MMU Assist Register Field Updates—Motorola Book E Standard

MAS Register Bit/Field	Value Loaded for Each Case						
	Instr/Data TLB Error	tlbsx Hit	tlbsx Miss	ISI	DSI	tlbre	tlbwe
NV	If TLBSELD supports NV: next NV (array) else, undefined	If TLBSEL supports NV: NV (array) else, undefined	If TLBSELD supports NV: next NV (array) else, undefined	—	—	If TLBSEL supports NV: NV (array) else, undefined	—
V	1	1	0	—	—	V (array)	—
IPROT	0	If TLB that hits supports IPROT: matched IPROT value; else 0	0	—	—	If TLB that hits supports IPROT: matched IPROT value; else 0	—
TID	Value of PID register selected by TIDSELD	TID (array)	SPID0	—	—	TID(array)	—
TS	MSR[IS/DS]	SAS	SAS	—	—	TS (array)	—
TSIZE[0–3]	TSIZED	TSIZE (array)	TSIZED	—	—	TSIZE (array)	—
EPN[32–51]	If TLBSELD has fixed page size; EPN of access else, undefined	EPN (array)	—	—	—	EPN (array)	—
X0, X1 WIMGE	X0D, X1D WIMGED	X0, X1 (array) WIMGE (array)	X0D, X1D WIMGED	—	—	X0, X1 (array) WIMGE (array)	—
RPN[32–51]	Zeroes	RPN(array)	Zeroes	—	—	RPN (array)	—
PERMIS	Zeroes	PERMIS (array)	Zeroes	—	—	PERMIS (array)	—
TLBSELD	—	—	—	—	—	—	—
TIDSELD	—	—	—	—	—	—	—
TSIZED	—	—	—	—	—	—	—
WIMGED	—	—	—	—	—	—	—

11.3.7 TLB Configuration Information

Information about the configuration for a given TLB implementation is available to system software by reading the contents of the MMU configuration SPRs. These SPRs describe the architectural version of the MMU, the number of TLB arrays, and the characteristics of each TLB array. MMU architecture version number 1 is defined as these SPRs with the field definitions as described in this section.

11.3.7.1 MMU Configuration Register (MMUCFG)

The MMU configuration register (MMUCFG) must be implemented by all Motorola Book E processors. It contains basic information about the MMU architecture for this device. MMUCFG contents can be read using an **mfspr** instruction. Figure 11-20 shows the format for MMUCFG.

	32	48	49	52	53	57	58	59	60	61	62	63
Field	—			NPIDS		PIDSIZE		—	NTLBS		MAVN	
Reset	All zeros											
R/W	Read only											
SPR	SPR1015											

Figure 11-20. MMU Configuration Register (MMUCFG) Format

The fields of MMUCFG are described in Table 11-16.

Table 11-16. MMUCFG Field Descriptions

	Bits	Name	Description
Motorola Implementation Standard	32–48	—	Reserved, should be cleared.
	49–52	NPIDS	Number of PID registers. A 4-bit field that indicates the number of PID registers provided by the processor.
	53–57	PIDSIZE	PID register size. The 5-bit value of PIDSIZE is one less than the number of bits in each of the PID registers implemented by the processor. The processor implements only the least significant PIDSIZE+1 bits in the PID registers.
	58–59	—	Reserved, should be cleared.
	60–61	NTLBS	Number of TLBs. The value of NTLBS is one less than the number of software-accessible TLB structures that are implemented by the processor. NOTE: NTLBS is set to one less than the number of TLB structures so that its value matches the maximum value of MAS0[TLBSEL].
	62–63	MAVN	MMU architecture version number. Indicates the version number of the architecture of the MMU implemented by the processor. 0b00 indicates version 1.0.

11.3.7.2 TLB Configuration Registers (TLBnCFG)

TLB configuration registers *n* (TLBnCFG), must be implemented by all Motorola Book E processors for each of the TLBs specified in MMUCFG[NTLBS]. They contain configuration information about each particular TLB.

Figure 11-21 shows the format for TLB_nCFG.

	32	39	40	43	44	47	48	49	50	51	52	63	
Field	ASSOC			MINSIZE		MAXSIZE		IPROT	AVAIL	—		NENTRY	
Reset	Implementation specific (see Section 2.12.4, "TLB Configuration Registers (TLB _n CFG)")												
R/W	R/W												
SPR	TLB0CFG: SPR688; TLB1CFG: SPR689												

Figure 11-21. TLB Configuration Registers (TLB_nCFG) Format

TLB_nCFG are defined as 32-bit read-only registers. TLB_nCFG bits are numbered 32 (most-significant bit) to 63 (least-significant bit). The contents of the TLB_nCFG registers can be read into bits 32–63 of a GPR using an **mf spr rT**, TLB_nCFG instruction.

The TLB configuration register number assignment is the same as the value in MAS0[TLBSEL]. For example, TLB0CFG register provides configuration information about TLB0, and TLB1CFG register provides configuration information about TLB1.

The fields of TLB_nCFG are described in Table 11-17.

Table 11-17. TLB Configuration Register *n* (TLB_nCFG) Field Descriptions

	Bits	Name	Description
Motorola Implementation Standard	32–39	ASSOC	Associativity. Specifies TLB associativity. If ASSOC equals the value in TLB _n CFG[NENTRY], the TLB is fully associative.
	40–43	MINSIZE	Minimum page size. Specifies the smallest page size (4^{MINSIZE} Kbytes) supported by the TLB.
	44–47	MAXSIZE	Maximum page size. Specifies the largest page size (4^{MAXSIZE} Kbytes) supported by the TLB.
	48	IPROT	Invalidate protect capability 0 Invalidate protection capability is not supported by this TLB. 1 Invalidate protection capability is supported by this TLB.
	49	AVAIL	Page size availability 0 All page sizes of 4^{SIZE} KB where $\text{SIZE} \in \{\text{MINSIZE}, \dots, \text{MAXSIZE}\}$ are supported. 1 The TLB supports page sizes of 4^{MINSIZE} and 4^{MAXSIZE} only. If MINSIZE = MAXSIZE, writing either value to this field is valid and the implementation clears AVAIL.
	50–51	—	Reserved, should be cleared.
	52–63	NENTRY	Number of entries. Specifies the number of entries supported by the TLB. Note that a value of zero is possible to indicate that the TLB array is empty and cannot be used.

Chapter 12

L1 Caches

The e500 core complex contains separate 32-Kbyte, eight-way set associative level 1 (L1) instruction and data caches to provide the execution units and registers rapid access to instructions and data.

This chapter describes the organization of the on-chip L1 instruction and data caches, cache coherency protocols, cache control instructions, and various cache operations. It describes the interaction that occurs in the memory subsystem, which consists of the memory management unit (MMU), the caches, the load/store unit (LSU), and the core complex bus (CCB). This chapter also describes the replacement algorithms used for each of the L1 caches.

Note that in this chapter, the term ‘multiprocessor’ is used in the context of maintaining cache coherency. These multiprocessor devices could be actual processors or other devices that can access system memory, maintain their own caches, and function as bus masters requiring cache coherency.

12.1 Overview

The core complex L1 cache implementation has the following characteristics:

- It consists of two separate 32-Kbyte instruction and data caches (Harvard architecture).
- Instruction and data caches are eight-way set associative, non-blocking.
- Cache directories are physically addressed. The physical (real) address tag is stored in the cache directory.
- Two-cycle access time providing three-cycle read latency for instruction and data caches accesses; accesses pipelined providing single-cycle throughput from caches.
- Instruction and data caches have 32-byte cache blocks. A cache block is the block of memory that a coherency state describes, also referred to as a cache line.
- The data cache supports a four-state modified/exclusive/shared/invalid (MESI) or three-state (MEI) coherency protocol.
- Both L1 caches support parity generation and checking (enabled through bits in the L1CSR0 and L1CSR1 registers) as follows:

- Instruction cache—one parity bit per byte of instruction
- Data cache—one parity bit per byte of data
- Each cache can be independently invalidated through cache flash invalidate (CFI) control bits located in the L1CSR1 and L1CSR0 registers.
- The caches implement a pseudo-least-recently-used (PLRU) replacement algorithm
- Supports individual line locking

Bus snooping ensures the coherency of global memory with respect to the data cache.

Both instruction and data cache lines are filled in a single-cycle 32-byte write from a line fill buffers described in Section 12.1.1.1, “Load/Store Unit (LSU),” and Section 12.1.1.2, “Instruction Unit Cache.” Cache line fills write all 32 bytes at once, and therefore do not occur until all four data beats of 8 bytes each have been loaded into the line fill buffer from the core complex bus (CCB).

Both instruction and data accesses are performed critical double word first on the CCB. For data accesses, the load/store unit (LSU) receives the critical double word as soon as it is available; it does not wait for all 32 bytes. That data is then forwarded to the requesting unit before being written to the cache, thus minimizing stalls due to cache fill latency. For instruction accesses, instruction fetching cannot resume until the entire cache line is loaded in the instruction line fill buffer (ILFB). Then, the critical double word is written to the cache and instruction fetching can resume.

12.1.1 Block Diagram

The instruction and data caches are integrated with the LSU, instruction unit, and CIU in the memory subsystem of the core complex as shown in Figure 12-1.

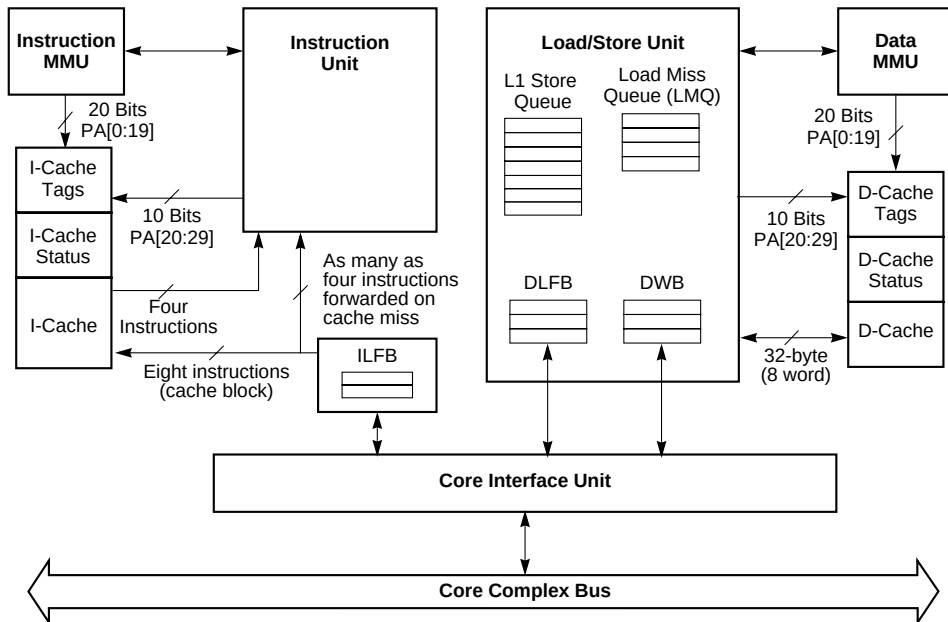


Figure 12-1. Cache/Core Interface Unit Integration

The following sections briefly describe the LSU, the instructions unit, the core interface unit and the core complex bus.

12.1.1.1 Load/Store Unit (LSU)

The data cache supplies data to the general-purpose registers (GPRs) by means of the load/store unit (LSU). The core complex LSU is directly coupled to the data cache with a 32-byte interface (the width of a cache block) to allow efficient movement of data to and from the GPRs. The LSU provides all of the logic required to calculate effective addresses, handles data alignment to and from the data cache, provides sequencing for load/store multiple operations, and interfaces with the core interface unit. Write operations to the data cache can be performed on a byte, half-word, word, or double-word basis.

This section describes the LSU queues that support the L1 data cache. See Section 12.3.5, “Load/Store Operations,” for more information on architectural coherency implications of load/store operations and the LSU on the core complex. Also, see Section 4.4.5, “Load/Store Unit Execution Timing,” for more information on other aspects of the LSU and instruction scheduling considerations.

12.1.1.1.1 Caching-allowed Loads and LSU

When free of data dependencies, caching-allowed loads execute in the LSU in a speculative manner with a maximum throughput of one per cycle and a total 3-cycle latency for integer loads. Data returned from the cache on a load is held in a rename buffer until the completion logic commits the value to the processor state.

12.1.1.1.2 LSU Store Queue

Stores cannot be executed speculatively and are held in the seven-entry store queue, shown in Figure 12-1, until the completion logic indicates that the store instruction is to be committed. The store queue arbitrates for access to the L1 data cache. When arbitration is successful, the data is written to the data cache and the store is removed from the store queue. If a store is caching-inhibited, the operation moves through the store queue on to the rest of the memory subsystem.

12.1.1.1.3 LSU L1 Load Miss Queue (LMQ)

As loads come to the LSU, the LSU tries to access the cache. If there is a hit, the cache returns the data. If there is a miss, the LSU allocates an entry in the four-entry load miss queue (LMQ) and the three-entry data line fill buffer (DLFB), see Section 12.1.1.1.4, “LSU Data Line Fill Buffer (DLFB),” . The LSU then queues a bus transaction to read the line. If a subsequent load hits, the cache returns the results. If a subsequent load misses, the LSU allocates a second entry in the LMQ and, if the load is to a different cache line than the already outstanding miss, the LSU allocates the second entry of the DLFB and queues a second read transaction on the bus. If the load miss is to the same cache line as the already outstanding miss, it is not necessary for the LSU to allocate the second entry in DLFB.

The LSU continues processing load hits and load misses until one of the following conditions occurs:

- The four-entry LMQ is full and there is a fifth load miss.
- The LSU tries to perform a load miss, all three of the line fill entries in the DLFB are full, and the load is not to any of the cache lines that are represented in the DLFB.

12.1.1.1.4 LSU Data Line Fill Buffer (DLFB)

The data line fill buffer (DLFB) is located in the LSU. The three entries in the DLFB are used for loads and caching-allowed stores. Stores are allocated in the DLFB so that loads can access data from the store immediately (loads cannot access data from the L1 store queue). Also, by using the entries in the DLFB for stores, the LSU frees up entries in the L1 store queue. Multiple caching-allowed store misses are merged in DLFB. See Section 12.6.1.4, “Store Miss Merging,” for more information.

12.1.1.1.5 LSU Data Write Buffer (DWB)

When a full line of data is available in the DLFB, the data cache is updated. If a data cache update requires that a line currently in the cache be evicted, that line is castout and placed in the Data Write Buffer (DWB) until the data has been transferred through the CIU to the CCB. If global memory's coherency needs to be maintained, as a result of bus snooping, the L1 cache can also evict a line to the DWB. This write is called a snoop push operation. Note that all castout and snoop push writes from the L1 cache are cache-line aligned (critical word is not written first). This is independent of which word in a modified cache line is accessed.

There are three entries in the DWB: one dedicated for snoop pushes, one dedicated for castouts, and one that can be used for either snoop pushes or castouts.

12.1.1.2 Instruction Unit

The instruction unit interfaces with the L1 instruction cache and the core interface unit. When instructions miss in the instruction cache they are accumulated in the instruction line fill buffer (ILFB) as they are fetched. After an entire line is available, it is written into the instruction cache and the ILFB is emptied.

12.1.1.3 Core Interface Unit

The core interface unit (CIU) handles all bus transactions initiated by the ILFB, DLFB, and DWB. The CIU handles all ordering and bus protocol and is the interface between the core complex and the external memory and caches.

The CIU performs transactions through the CCB by transferring either the critical double word first (8 bytes) or the critical quad word first (16 bytes). It then forwards the transaction to the instruction or data line fill buffer critical double word first. The CCB also captures snoop addresses for the L1 data cache and the memory reservation (**lwarx** and **stwcx**) operations.

12.2 L1 Cache Organization

The L1 instruction and data caches of the core complex are both organized as 128 sets of eight blocks with 32 bytes in each cache line. The following subsections describe the differences in the organization of the instruction and data caches.

12.2.1 L1 Data Cache Organization

The L1 data cache is organized as shown in Figure 12-2.

L1 Cache Organization

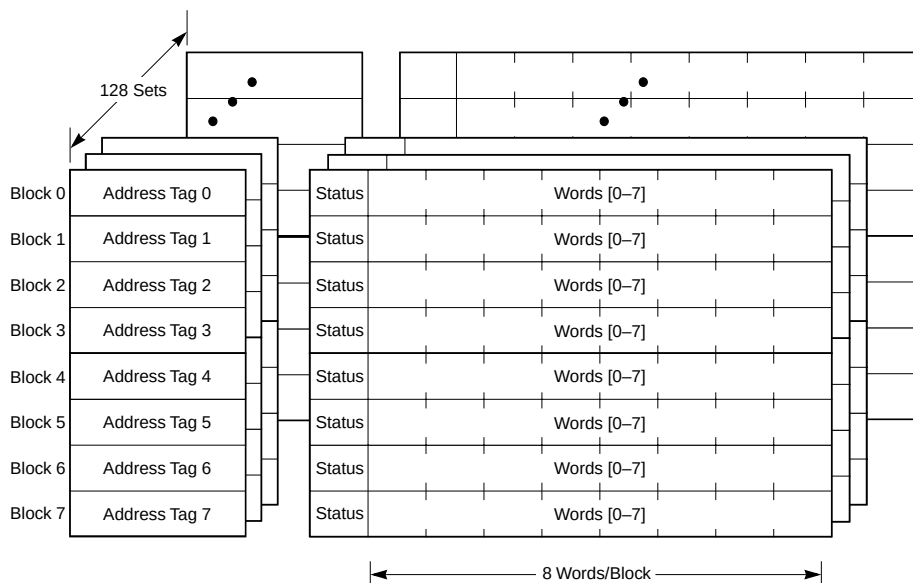


Figure 12-2. L1 Data Cache Organization

Each block consists of 32 bytes of data, three status bits, one lock bit, and an address tag. For the L1 data cache, a cache block is the 32-byte cache line. Also, although it is not shown in Figure 12-2, the data cache has one parity bit/byte (four parity bits/word).

Each cache block contains 8 contiguous words from memory that are loaded from an 8-word boundary (that is, bits 27-31 of the physical addresses are zero). Cache blocks are also aligned on page boundaries. Physical address bits PA[20:26] provide the index to select a cache set. The tags consist of physical address bits PA[0:19]. Address translation occurs in parallel with set selection (from PA[20:26]). Lower address bits PA[27:31] locate a byte within the selected block.

The data cache can be accessed internally while a fill for a miss is pending (allowing hits under misses) and the data from a hit can be used as soon as it is available. The LSU forwards the critical word to any pending load misses and allows them to finish. Later, when all the data for the miss has arrived, the entire cache line is reloaded. In addition, subsequent misses can also be sent to the memory subsystem before the original miss is serviced (allowing misses under misses). Up to four misses can be pending in the load miss queue. See Section 12.1.1.1.3, “LSU L1 Load Miss Queue (LMQ),” for more information.

There are three status bits associated with each cache block. These bits are used to implement the modified/exclusive/shared/invalid (MESI) cache coherency protocol. The coherency protocols are described in Section 12.3, “Cache Coherency Support.”

12.2.2 L1 Instruction Cache Organization

The L1 instruction cache is organized as shown in Figure 12-3.

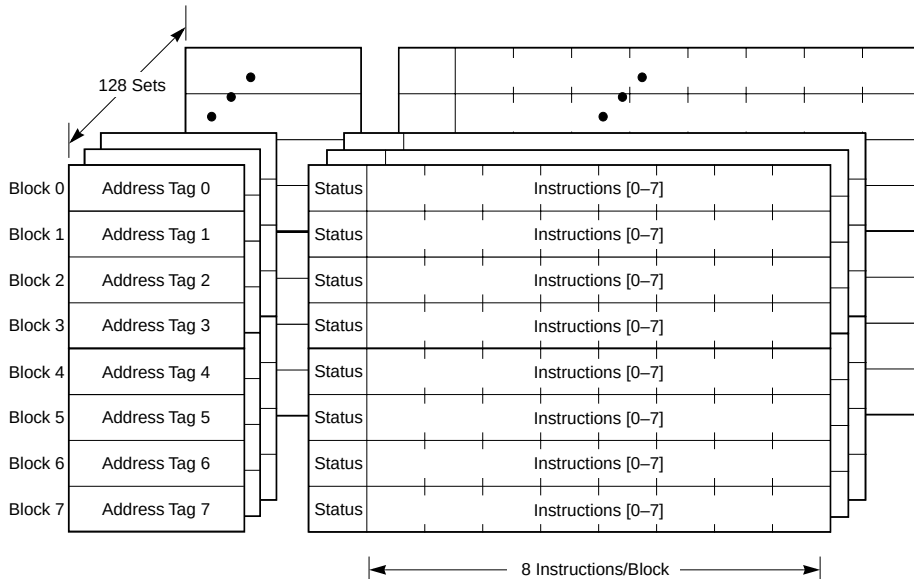


Figure 12-3. L1 Instruction Cache Organization

Each block consists of eight instructions, one status bits, one lock bit, and an address tag. Also, although it is not shown in Figure 12-3, the instruction cache has one parity bit/byte, yielding 32 parity bits for each line.

As with the data cache, each instruction cache block is loaded from an 8-word boundary (that is, bits 27-31 of the physical addresses are zero). Instruction cache blocks are also aligned on page boundaries. Also, PA[20:26] provides the index to select a set, and PA[27:28] selects an instruction within a block. The tags consist of physical address bits PA[0:19]. Address translation occurs in parallel with set selection (from PA[20:26]).

The instruction cache can be accessed internally while a fill for a miss is pending (allowing hits under misses). Although the data cannot be used, the hit information stops a subsequent miss from requesting a fill. In addition, subsequent misses can also be sent to the memory subsystem before the original miss is serviced (allowing misses under misses). When a miss is actually updating the cache, subsequent accesses are blocked for 1 cycle (but up to four instructions being loaded into the instruction cache can be forwarded to the instruction unit simultaneously).

The instruction cache differs from the data cache in that it does not implement a multiple-state cache coherency protocol. A single status bit indicates whether a cache block is valid or invalid and there is a single bit for locking.

12.3 Cache Coherency Support

This section describes the L1 data and instruction cache coherency models and coherency support.

12.3.1 Data Cache Coherency Model

The core complex data cache supports either a three-state or a four-state cache coherency protocol for cache lines in the data cache. The three-state protocol (also referred to as MEI protocol) consists of the coherency states invalid, exclusive, and modified which is optimal for infrequently shared pages. The three-state protocol is hardware coherent across the CCB; however it is not optimized for efficient and frequent sharing of data between bus masters. (This protocol is also implemented by earlier MPC603 and MPC750 processors.)

The four-state protocol (also referred to as MESI protocol) includes the additional shared state. This protocol supports efficient and frequent sharing of data between bus masters. However, out of reset, the core complex defaults to the three-state coherency protocol by performing all burst reads as read-with-intent-to-modify (in order to flush the line from other caches). In order to change the core complex into a four-state coherency protocol, the HID1 register must be changed in one of two ways. Either HID1[MSHARS] must be set and the SHAREN bit in the corresponding TLB entry must also be set (HID1[SSHAR] is ignored), or HID1[MSHARS] must remain cleared and HID1[SSHAR] must be set. If HID1[MSHARS] is cleared, the value of the TLB entries SHAREN bit is ignored, and three-state and four-state protocol is determined solely by the value in HID1[SSHAR]. Table 12-1 describes the conditions which set the coherency state protocol. See Section 13.3.2.2, “SHAREN Page Attribute,” for more information.

Table 12-1. Coherency Protocol

HID1[MSHAR]	HID1[SSHAR]	TLB Entries SHAREN Bit	Coherency State Protocol
0	0	Ignored	MEI
0	1	Ignored	MESI
1	Ignored	0	MEI (determined on a page by page basis by TLB entry SHAREN bit)
1	Ignored	1	MESI (determined on a page by page basis by TLB entry SHAREN bit)

In the four-state coherency protocol all burst reads are performed as read-only transactions, possibly writing the line into the cache as shared. Stores that hit on shared lines in the data cache invalidate the line, and fetch the line from memory as read-with-intent-to-modify.

Each 32-byte data cache block contains three status bits (valid, shared, and modified) that define the MESI state of the cache line. The core complex uses these bits to support the coherency protocols and to direct reload operations. The L1 data cache states are defined in

Table 12-2. In MEI mode, the same state definitions apply except for the shared state definition.

Table 12-2. Cache Line State Definitions

MESI [0–2]	Name	Description
101	Modified (M)	The line is present in the cache, and it has been modified with respect to main memory. The line does not reside in any other coherent caches.
100	Exclusive (E)	The addressed line is present in the cache, and this cache has exclusive ownership of the line. It is not present in any other coherent cache and it is the same as main memory. This processor may subsequently modify this line without notifying other bus masters.
110	Shared (S)	The addressed line is present in the cache, and it may be also present in another coherent cache and it is the same as main memory. It can not be modified by any processor.
0xx	Invalid (I)	The cache location does not contain valid data.

Every data cache block state is defined by its MESI status bits. Note that in a multiprocessor system, a cache line can exist in the exclusive state in at most one L1 data cache at any one time.

Table 12-3 describes how execution of instructions affects the coherency states of the L1 data cache. Table 12-3 also takes into account the settings of the WIM bits. For more information on the WIM bits, see Section 12.3.4, “WIMGE.”

Table 12-3. L1 Data Cache Coherency State Transitions

Event	WIM	Initial State	Final State
dcba	00x ¹	Any	M
dcbf	xxx	Any	I
dcbi	xxx	Any	I
dcblc (CT = 0)	xxx	Any	same
dcblc (CT = 1)	xxx	Any	same
dcbst	xxx	Any	I
dcbt (CT = 0)	x0x	M, E, or S	same
dcbt (CT = 0) (MESI only)	x0x	I	S or E
dcbt (CT = 0) (MEI only)	x0x	I	E
dcbt (CT = 1)	x0x	Any	I
dcbtls (CT = 0)	x0x	M, E, or S	same
dcbtls (CT = 0) (MESI only)	x0x	I	S or E
dcbtls (CT = 0) (MEI only)	x0x	I	E
dcbtls (CT = 1)	x0x	Any	I
dcbtst (CT = 0)	00x	M, E, or S	same

Table 12-3. L1 Data Cache Coherency State Transitions

Event	WIM	Initial State	Final State
dcbtst (CT = 0)	00x	I	E
dcbtst (CT = 1)	00x	Any	I
dcbtstls (CT = 0)	00x	M or E	same
dcbtstls (CT = 0)	00x	S or I	E
dcbtstls (CT = 1)	00x	Any	I
dcbz	00x	Any	M
icblc (CT = 1)	xxx	Any	same
icbt (CT = 1)	x0x	Any	I
icbtls (CT = 1)	x0x	Any	I
Load	xxx	M, E, or S	same
Load (MESI only)	x0x	I	S or E
Load (MEI only)	x0x	I	E
Load	x1x	Any	same
lwarx	00x	M, E, or S	same
lwarx (MESI only)	00x	I	S or E
lwarx (MEI only)	00x	I	E
lwarx	01x	Any	same
Store	00x	Any	M
Store	10x	M or E	same
Store	10x	S or I	I
Store	01x	Any	same
stwcx	00x	Any	M

¹ The x indicates that the value is either 0 or 1

The core complex provides full hardware support for PowerPC cache coherency and ordering instructions and full hardware implementation of the TLB management instructions. The core complex broadcasts cache management instructions (**dcbst**, **dcbic** (CT = 1), **icblc** (CT = 1), **dcbf**, **dcbi**, **mbar**, **msync**, **tlbi**, **tlbsync**, **icbi**) based on the address broadcast enable (ABE) bit in HID1 to allow management of external L2 caches (for implementations with L2 caches) as well as other L1 caches in the system.

12.3.1.1 Shared Memory Multiprocessor Support

For optimum uniprocessor performance, all lines should be fetched from memory as read-with-intent-to-modify by setting the page for three-state MEI mode. Multiprocessor systems can be set to three-state mode as described in Table 12-1.

12.3.2 Instruction Cache Coherency Model

The instruction cache does not implement the three- and four-state cache coherency protocols of the data cache. The only states for a cache line in the instruction cache are invalid and valid. Table 12-4 describes how execution of instruction cache control instructions affect the coherency states of the L1 instruction cache.

Table 12-4. L1 Instruction Cache Coherency State Transitions

Event	WIM	Initial State	Final State
icbi	xxx	V or I	I
icblc (CT = 0)	xxx	V or I	same
icbtlis (CT = 0)	x01	V or I	V

The instruction cache is loaded only as a result of instruction fetching or by an Instruction Cache Block Touch and Lock Set (**icbtlis**) instruction. It is not snooped for general coherency with other caches; however, it is snooped when the Instruction Cache Block Invalidate (**icbi**) instruction is executed by this processor or any other processor in the system. Instruction cache coherency must be maintained by software and is supported by a fast hardware flash invalidation capability as described in Section 12.5, “L1 Data Cache Flushing.” Also, the flushing of self-modifying code from the data cache is described in Section 11.1.2.1, “Self-Modifying Code.”

12.3.3 Snoop Signaling

Cache coherency is maintained automatically by hardware through snooping on the CCB. A bus transaction is enabled for snooping by setting the coherency-required bit (M) in the TLBs (WIMGE = 0bxx1xx). The state of this bit is transmitted with the address on the CCB on the internal global pin (ccb_gbl). When the internal ccb_gbl is asserted, the CCB transaction should be snooped by other bus masters.

The action to take according to the cache coherency protocol due to a snoop is implicitly indicated by the transfer type (TTx) encodings, which are transmitted on the CCB with the address. These encodings indicate whether a transaction is a read or write and whether a reading bus master has an intent to modify the cache line. The core complex uses these encodings as a CCB master to signal its intent to other snooping caches.

The three-state/four-state mode selection for a page does not change the interpretation of snoops. Clean snoop transactions transition a cache line to the shared state even if the core complex is operating the page in three-state (MEI) mode. In this case, the line operates in the four-state coherency model until the core complex performs an operation that normally puts the line back into the exclusive or modified state. This transition, if it occurs, takes place normally according to the four-state coherency model.

Clean, flush, and kill are three basic snoops that affect the L1 data cache. The state changes caused by these snoops are described in Table 12-5.

The instruction cache is not snooped, except in the case of the ikill, so coherency must be maintained by software. However, the core complex does support a fast instruction cache invalidation capability as described in Section 12.4.3, “L1 Instruction and Data Cache Flash Invalidation.” Also, the flushing of self-modifying code from the data cache is described in Section 11.1.2.1, “Self-Modifying Code.”

For a description of the state changes caused by the ikill snoop, see Table 12-6.

Table 12-5. Data Cache Snoop Coherency State Transitions

Event	Initial State	Final State
clean	M, E, or S	S
clean	I	I
flush	Any	I
kill	Any	I

Table 12-6. Instruction Cache Snoop Coherency State Transitions

Event	Initial State	Final State
ikill	V or I	I

12.3.4 WIMGE

All instruction and data accesses are performed under the control of the WIMGE bits. This section describes how the settings of these bits affects the behavior of the L1 caches. For more information about the WIMGE bits, see Section 11.1.1, “Memory/Cache Access Attributes (WIMGE Bits).”

12.3.4.1 Write-Back Stores

If a write-back store hits to a line that is already in the exclusive state, it is immediately stored to and the state is changed to modified. If a write-back store hits to a line that is already in the modified state, it is immediately stored to and stays as modified.

If a write-back store operation (that is, caching-allowed and not write-through) hits a line in the shared state, the cache line is invalidated and a burst read-with-intent-to-modify is issued to the core complex bus. The line is received through the CIU and stored into the cache modified state.

12.3.4.2 Write-Through Stores

A write-through store operation (WIMGE = 0b10xxx) may hit an exclusive cache line. In this case, the store data is written into the data cache and the write-through store goes to the core complex bus as a single-beat write. The cache line stays exclusive.

A write-through store operation may hit in a shared cache line. In this case, that cache line is invalidated from the cache, and the write-through store goes to the core complex bus as a single-beat write.

A write-through store may also hit in a cache line that is already in the modified state. This situation normally occurs as a result of page table aliasing in which two effective addresses are mapped to the same physical page, but with one mapped as write-through and the other mapped as write-back (that is, non-write-through). In this case, the cache line remains in its current state, the store data is written into the data cache, and the store goes to the core complex bus as a single-beat write.

12.3.4.3 Caching-Inhibited Loads and Stores

A caching-inhibited load or store (WIMGE = 0bx1xxx) that hits in the cache presents a cache coherency paradox and is normally considered a programming error. If a caching-inhibited load hits in the cache, the cache data is ignored and the load is provided from the core complex bus as a single-beat read. If a caching-inhibited store hits in the cache, the cache may be altered but the store is performed on the core complex bus anyway as a single beat write.

12.3.4.4 Misaligned Accesses and the Endian (E) Bit

The following problem can be encountered within a single processor:

- Misaligned accesses across pages with different endianness

Misaligned load or store accesses that cross page boundaries could cause data corruption if the two pages are not set to have the same endianness (that is, one page is big endian while the other page is little endian) and the access is allowed. When this situation occurs, the core complex takes a DSI exception and sets the BO (byte ordering) bit in the exception syndrome register (ESR) instead of performing the accesses.

12.3.4.5 Speculative Accesses to Guarded Memory

There is no restriction on how the core complex performs instruction fetching from guarded memory, if the memory area is marked as execute-permitted (UX/SX = 1) in the TLBs. Note that software should mark guarded space as no-execute (UX = 0 and SX = 0) to prevent inadvertent instruction fetching from guarded areas of memory. Then, if the effective address of the current instruction is in guarded, no-execute memory, an execute access control exception occurs, generating an instruction storage interrupt.

The core complex does not perform speculative stores to guarded memory. However, loads from guarded memory may be accessed speculatively if one of the following applies:

- The target location is valid in the data cache.
- The load is guaranteed to be executed. In this case, the entire cache block containing the referenced data may be loaded into the cache.

For more information, see Section 11.1.1.4, “Guarded Attribute.”

12.3.5 Load/Store Operations

Load and store operations are assumed to be weakly ordered on the core complex. The load/store unit (LSU) can perform load operations that occur later in the program ahead of store operations, even when the data cache is disabled (see Section 12.3.5.2, “Sequential Consistency of Memory Accesses”).

12.3.5.1 Performed Loads and Stores

The architecture defines a performed load operation as one that has the addressed memory location bound to the target register of the load instruction. The architecture defines a performed store operation as one where the stored value is the value that any other processor will receive when executing a load operation (that is, of course, until it is changed again). With respect to the core complex, caching-allowed (WIMGE = 0bx0xxx) loads and caching-allowed, write-back (WIMGE = 0b00xxx) stores are performed when they have arbitrated to address the cache block in the L1 data cache or the CCB and therefore gained coherency ownership of the cache line (that is, they have gained M, E, or S (if enabled) rights to the line). The e500 considers caching-inhibited (WIMGE = 0bx1xxx) loads and stores, and write-through (WIMGE = 0b10xxx) stores performed when they have been successfully presented onto the CCB. Note that loads are considered performed at the L1 data cache only if the respective cache contains a valid copy of that address. Write-back stores are considered performed at the L1 data cache only if the respective cache contains a valid, non-shared copy of that address.

12.3.5.2 Sequential Consistency of Memory Accesses

The architecture requires that all memory operations executed by a single processor be sequentially consistent with respect to that processor as described in Section 11.1.3.1, “Memory Access Ordering.” This means that memory accesses appear to occur in program order with respect to exceptions and data dependencies.

The core complex achieves sequential consistency by operating a single data pipeline to the cache/MMU. Therefore, all memory accesses are presented to the MMU in program order and exceptions are determined in order. Loads are allowed to bypass stores after exception checking has been performed for the store, but data dependency checking is handled in the load/store unit so that a load does not bypass a store with an address match. Newer

caching-allowed loads can bypass older caching-allowed loads only if the two loads are to different 32-byte address granules. Newer caching-allowed write-back stores can bypass older caching-allowed write-back stores if they do not store to overlapping bytes of data.

Note that although memory accesses that miss in the L1 cache are forwarded onto the CIU for future arbitration onto the CCB, all potential synchronous exceptions are resolved before the cache access. In addition, although subsequent memory accesses can address the cache, full coherency checking between the cache and the CIU is provided to avoid dependency conflicts.

12.3.5.3 Enforcing Store Ordering with Respect to Loads

The e500 core complex guarantees that any load followed by any store is performed in order (with respect to each other). The reverse, however, is not guaranteed. An **mbar** instruction must be inserted between a store followed by a load to ensure sequential ordering between that store and that load.

12.3.5.4 Atomic Memory References

The core complex implements **lwarx** and **stwcx**, as described in Book E and in Section 11.1.3.3, “Atomic Memory References.” If the EA is not a multiple of 4 for either instruction, an alignment interrupt is invoked. Executing **lwarx** or **stwcx** to areas marked write-through causes a DSI exception.

As specified in Book E, the core complex requires that, for **stwcx**, to succeed, the EA of **stwcx** must be to the same reservation granule as the EA of a preceding **lwarx**. The core complex makes reservations on behalf of aligned 32-byte blocks of the memory address space.

If the reservation has been cancelled for any reason, then **stwcx** fails and clears CR0[EQ]. The architectural intent is to follow the **lwarx/stwcx** instruction pair with a conditional branch that checks whether **stwcx** failed.

The state of the reservation coherency bit is always presented onto the `ccb_rsrv` output. This can be used to determine when an internal condition caused the coherency bit to be reset.

The reservation is invalidated when any asynchronous interrupt is signaled. An external interrupt or watchdog timer are examples of asynchronous interrupts.

12.4 L1 Cache Control

The core complex L1 caches are controlled by programming specific L1CSR_n bits and by issuing dedicated cache control instructions. Section 12.4.1, “Cache Control Instructions,” describes the cache control instructions and gives implementation specific information. The remainder of this section describes how the cache control instructions and the L1CSR_n bits are used to control the L1 cache.

12.4.1 Cache Control Instructions

The following instructions can be used for management of the L1 caches in the e500 core—**dcba**, **dcbf**, **dcbi**, **dcblc**, **dcbst**, **dcbt**, **dcbtls**, **dcbstst**, **dcbststls**, **dcbz**, **icbi**, **icblc**, **icbt**, and **icbtls**.

Table 12-7 shows how the cache control instructions apply to the e500 core, Book E architecture, and the AIM definition of the PowerPC architecture.

Table 12-7. Cache Instruction Comparison

Instruction	e500 Core	Book E	PowerPC Architecture
dcba	x	x	x
dcbae		x	
dcbf	x	x	x
dcbfte		x	
dcbi	x	x	x
dcbie		x	
dcblc	x		
dcbst	mapped to dcbf	x	x
dcbstte		x	
dcbt	x	x	x
dcbte		x	
dcbtls	x		
dcbstst	x	x	x
dcbstste		x	
dcbststls	x		
dcbz	x	x	x
dcbze		x	
icbi	x	x	x
icbie		x	
icblc	x		
icbt	no-op	x	
icbte		x	
icbtls	x		

If a cache instruction causes multiple no-op or exception conditions, the results are determined by the order of precedence described in Table 12-8. The priority of the

conditions decreases from left to right and the dashes indicate that the operation executes normally.

Table 12-8. Failed Cache Events

Operation	MMU Miss	User and MSR[UCL E] = 0	Protection Violation	CT=0, CE=0	CT ≠ 0 or 1	CI	WT
dcbt dcbtst	no-op no-op	— —	no-op no-op	— —	no-op no-op	no-op no-op	— no-op
dcbtls dcbtstls dcblc	DTLB DTLB DTLB	DLK DLK DLK	DSI DSI DSI	CUL CUL no-op	CUL CUL no-op	CUL CUL —	— CUL —
icbtls icblc	DTLB DTLB	ILK ILK	DSI DSI	CUL no-op	CUL no-op	CUL —	— —
dcbz dcba	DTLB no-op	— —	DSI no-op	— —	— —	ALI no-op	ALI no-op
dcbf dcbi icbi	DTLB DTLB DTLB	— — —	DSI DSI DSI	— — —	— — —	— — —	— — —
lwarx stwcx.	DTLB DTLB	— —	DSI DSI	— —	— —	— —	DSI DSI
load store	DTLB DTLB	— —	DSI DSI	— —	— —	— —	— —

Note that CE describes the cache enable bit in L1CSR1 (for the instruction cache) or L1CSR0 (for the data cache). DLK and ILK indicate that the condition causes a data storage interrupt and sets the ESR[DLK] or ESR[ILK]. CUL indicates the unable-to-lock condition that results in a no-op and sets the CUL bit in L1CSR1 (for the instruction cache) or L1CSR0 (for the data cache).

Acronyms are used to signify the following interrupts:

- DTLB (data TLB interrupt)
- ALI (alignment interrupt)
- DSI (data storage interrupt)

All cache control instructions except **dcba**, **dcbt**, and **dcbtst** generate TLB miss exceptions if the effective address cannot be translated. The **dcba**, **dcbt**, and **dcbtst** instructions are treated as no-ops if the address cannot be translated.

If a **dcbt** or **dcbtst** instruction accesses a page marked caching-inhibited, it is treated as a no-op. The **icbt** instruction is treated as a no-op when the CT operand is equal to zero. The **dcbst** instruction maps to **dcbf**.

The core complex broadcasts the cache control instructions according to the value of HID1[ABE]. If ABE is cleared, most cache control instructions are not broadcast. If it is set, cache control instructions are broadcast.

12.4.2 L1 Instruction and Data Cache Enabling/Disabling

The instruction and data caches are enabled and disabled with the cache enable (CE) bits in L1CSR1 and L1CSR0, respectively. Disabling a cache does not cause all memory accesses to be performed as caching inhibited. When caching inhibited accesses are desired, the pages must be marked as caching inhibited in the MMU pages.

When either the instruction or data cache is disabled, the cache tag state bits are ignored and the corresponding cache is not accessed. The default power-up state of L1CSR0[CE] and L1CSR1[CE] is zero (caches disabled).

When the data cache is disabled, snooping of lines in the cache is not performed. Before the data cache is disabled it must be invalidated to prevent coherency problems when it is enabled again.

All cache operations are affected by disabling the cache. Touch instructions (**dcbt**, **dcbtst**, **dcblc**, **dcbtls**, **dcbtstls**, **icblc**, and **icbtls**) performed on the CCB by the e500 do not affect the cache when it is disabled. A **dcba** or **dcbz** instruction to a disabled data cache zeros the cache line in memory, but does not affect the cache when it is disabled.

The setting of L1CSR0[CE] must be preceded by an **msync** and **isync** instruction, to prevent a cache from being disabled or enabled in the middle of a data or instruction access. See Table 2-45 for more information on synchronization requirements.

12.4.3 L1 Instruction and Data Cache Flash Invalidation

When software guarantees that memory is not shared, the data cache can be invalidated by executing a series of **dcbi** instructions or by setting L1CSR0[CFI]. The instruction cache can be invalidated by setting L1CSR1[CFI]. The L1 caches can be flash invalidated independently. The setting of L1CSR0[CFI] and L1CSR1[CFI] must be preceded by an **msync** and **isync** instruction, respectively.

Both caches are invalidated automatically upon power-up. A subsequent reset operation does not invalidate the caches automatically. Therefore, software must use the CFI controls if invalidation is desired after a warm reset. Setting the CFI bits causes a flash invalidation performed in a single CPU cycle. After that single cycle, the CFI bits are automatically cleared (CFI bits are not sticky). Note that there is no broadcast of a flash invalidate operation on the CCB.

Individual instruction cache blocks and data cache blocks can be invalidated using the **icbi** and **dcbi** instructions, respectively. Note that with the **dcbi** instruction, the e500 core

invalidates the cache block without pushing it out to memory. See Section 3.4.1.9.1, “User-Level Cache Instructions.”

Exceptions and other events that can access the L1 cache should be disabled during this time so that the PLRU algorithm can function undisturbed.

12.4.4 L1 Instruction and Data Cache Line Locking/Unlocking

The core complex provides five user-mode instructions to perform cache line locking/unlocking based on the complete address of the cache line. Three of the instructions are for data cache locking and unlocking (**dcblic**, **dcbtlis**, and **dcbtstlis**) and the remaining instructions are for instruction cache locking (**icblic** and **icbtls**). For descriptions, see Section 3.11.5, “Cache Locking APU.”

The CT operand is used to indicate the cache target of the cache line locking instruction.

Lock instructions are treated as loads when translated by the data TLB, and they cause exceptions when data TLB errors or data storage interrupts occur.

The user-mode cache lock enable bit, MSR[UCLE], is used to restrict user-mode cache line locking by the operating system. If MSR[UCLE] = 0, any cache lock instruction executed in user mode (MSR[PR] = 1) causes a cache-locking DSI exception and sets either ESR[DLK] or ESR[ILK]. This allows the OS to manage and track the locking/unlocking of cache lines by user-mode tasks. If MSR[UCLE] is set, the cache-locking instructions can be executed in user-mode and do not cause a DSI for cache-locking. However, they may still cause a DSI for access violations.

If all of the ways are locked in a cache set, an attempt to lock another line in that set results in an overlocking situation. The new line is not placed in the cache, and either the data cache overlock bit L1CSR0[CLO] or instruction cache overlock bit L1CSR1[CLO] is set. This does not cause an exception condition.

The following cases cause an attempted lock to fail:

- The target address is marked caching-inhibited
- The corresponding cache is disabled and the CT operand of the cache locking instruction = 0
- The cache target operand(CT[6–10]) is greater than 1
- **dcbtstlis** is used for a target address of a write-through page

In these cases, the lock set instruction is treated as a no-op and the data cache unable-to-lock bit (L1CSR0[CUL]) or the instruction cache unable-to-lock bit (L1CSR1[CUL]) bit is set. This condition does not cause an exception.

It is acceptable to lock all ways of a cache set. A non-locking line fill for a new address in a completely locked cache set will not be put into the cache. It is, however, loaded into a DWB and creates the appropriate normal burst write transfer.

The cache-locking DSI handler must decide whether to lock a given cache line based on available cache resources.

If the locking instruction is a set lock instruction, to lock the line, the handler should do the following:

1. Add the line address to its list of locked lines.
2. Execute the appropriate set lock instruction to lock the cache line.
3. Modify save/restore register 0 (SRR0) to point to the instruction immediately after the locking instruction that caused the DSI.
4. Execute an **rfi**.

If the locking instruction is a clear lock instruction, to unlock the line, the handler should do the following:

1. Remove the line address from its list of locked lines.
2. Execute the appropriate clear lock instruction to unlock the cache line.
3. Modify SRR0 to point to the instruction immediately after the locking instruction that caused the DSI.
4. Execute an **rfi**.

Failure to update SRR0 to point to the instruction after the locking/unlocking instruction causes the exception handler to be repeatedly invoked for the same instruction.

12.4.4.1 Effects of Other Cache Instructions on Locked Lines

The following cache instructions have no effect on the state of a cache line's lock bit:

- **dcbt** (CT = 0)
- **dcbtst** (CT = 0)

If **dcbt** is performed to a line that is locked in the cache in the modified, exclusive, or shared (if enabled) state, **dcbt** takes no action. However, if the line is invalid, and therefore not locked, the **dcbt** instruction causes a state change.

If a **dcbtst** (CT=0) is performed to a line that is locked in the cache in the modified, exclusive, or shared (if enabled) state, **dcbtst** takes no action. If the line is invalid, and therefore not locked, the **dcbtst** instruction causes a state change.

The following cache instructions are treated as stores and may cause the invalidation and unlocking of a cache line in another processor in an MP system:

- **dcba**
- **dcbz**

In implementations with an L2 cache, the following instructions, when directed to the L2 cache (CT = 1), flush/invalidate and unlock a line in the L1 data cache of the current processor:

- **dcbt**
- **dcbtst**
- **dcbtls**
- **dcbtstls**
- **icbt**
- **icbtl**

The following cache instructions flush/invalidate and unlock a line in the cache of the current processor, and may also flush/invalidate and unlock a cache line in other processors in an MP system:

- **dcbf**
- **dcbst**
- **icbi**
- **dcbi**

12.4.4.2 Flash Clearing of Lock Bits

The core complex allows flash clearing of the instruction and data cache lock bits under software control. Each cache's lock bits can be independently flash cleared through the CLFC control bits in the L1CSR0 and L1CSR1 registers.

Lock bits in both caches are cleared automatically upon power-up. A subsequent reset operation does not clear the lock bits automatically. Software must use the CLFC controls if flash clearing of the lock bits is desired after a warm reset. Setting the CLFC bits causes a flash invalidation performed in a single CPU cycle. After that cycle, the CLFC bits are automatically cleared (CLFC bits are not sticky).

12.5 L1 Data Cache Flushing

The software flush routines flush the data cache by using the **dcbf** instruction or by executing a series of 12 uniquely addressed load or **dcbz** instructions to each of the 128 sets. The address space should not be shared with any other process to prevent snoop hit invalidations during the flushing routine. Exceptions should be disabled during this time so that the PLRU algorithm does not get disturbed.

12.6 L1 Cache Operation

This section describes the core complex cache operations performed by the L1 instruction and data caches.

12.6.1 Cache Miss and Reload Operations

This section describes the actions taken by the L1 caches on misses for caching-allowed accesses. Also, it describes what happens on cache misses for caching-inhibited accesses as well as disabled and locked L1 cache conditions.

12.6.1.1 Data Cache Fills

The core complex data cache blocks are filled (sometimes referred to as a cache reload) from an L2 cache or the memory subsystem when cache misses occur for caching-allowed accesses, as described in Section 12.1.1.1, “Load/Store Unit (LSU),” and Section 12.1.1.2, “Instruction Unit.”

When the data cache is disabled ($L1CSR0[CE] = 0$), data accesses bypass the data cache. These accesses are forwarded to the memory subsystem as caching-allowed and proceed to the CCB. When the data is returned, it is forwarded to the requesting execution unit, but it is not loaded into any of the caches.

Each of the 8 ways of each set in the data cache can be locked (by locking all of the cache lines in the way with the **dcbtls** or **dcbtstls** instruction). When at least one way is unlocked, misses are treated normally and they are allocated to one of the unlocked ways on a reload. If all 8 ways are locked, store/load misses proceed to the memory subsystem as normal caching-allowed accesses. In this case, the data is forwarded to the requesting execution unit when it returns, but it is not loaded into the data cache. If the data is modified, it is loaded into a DWB and creates the appropriate normal burst write transfer.

Each of the 8 ways of each set in the instruction cache can be locked (by locking all of the cache lines in the way with the **icbtl**s instruction). When at least one way is unlocked, misses are treated normally and they are allocated to one of the unlocked ways on a reload. If all 8 ways are locked, instruction misses proceed to the memory subsystem as normal caching-allowed accesses. In this case, the instruction is forwarded to the instruction unit when it returns, but it is not loaded into the instruction cache.

Note that caching-inhibited stores should not access any of the caches (see Section 12.3.4.3, “Caching-Inhibited Loads and Stores,” for more information). See Section 12.6.1.4, “Store Miss Merging,” for more information on the handling of caching-allowed store misses.

12.6.1.2 Instruction Cache Fills

The instruction cache provides a 128-bit interface to the instruction unit, so as many as four instructions can be made available to the instruction unit in a single clock cycle on an L1 instruction cache hit. On a miss, the core complex instruction cache blocks are loaded in one 32-byte beat from the CCB; the instruction cache is nonblocking, providing for hits under misses.

The instruction cache operates similarly to the data cache when all eight ways of a set are locked. When the instruction cache is disabled ($L1CSR1[CE] = 0$), instruction accesses bypass the instruction cache. These accesses are forwarded to the memory subsystem as caching-allowed and proceed to the CCB. When the instructions are returned, they are forwarded to the instruction unit but are not loaded into the instruction cache.

The instruction unit fetches a total of four instructions at a time directly from the memory subsystem for caching-inhibited instruction fetches. Similar to the data cache, when the instructions are returned, they are forwarded to the instruction unit but are not loaded into any of the caches in this case.

12.6.1.3 Cache Allocation on Misses

Instruction cache misses cause a new line to be allocated into the instruction cache on a PLRU basis, provided the cache is not completely locked or disabled.

If there is a data cache miss for a caching-allowed load or store (including touch instructions) and the line is not already going to be allocated into the data cache as a result of a previous load/store miss, the miss will cause a new line to be allocated into the data cache on a PLRU basis, provided the cache is not completely locked or disabled. A store that is write-through or caching-inhibited that misses in the data cache does not cause a fill. Also, cache ops (**dcbi**, **dcbf**, etc.) that miss in the cache do not cause a fill.

12.6.1.4 Store Miss Merging

When a caching-allowed store misses in the data cache, an entry is allocated in the DLFB. The store data is written into the DLFB. The remainder of the bytes not written by the store data are filled in when the cache block is eventually fetched from memory through the CCB. When all 32 bytes are valid, the cache block is reloaded into the data cache.

If a subsequent store miss hits on a DLFB entry for a previous store miss, the subsequent store miss also writes its data into the DLFB for that entry. Any number of stores that hit the DLFB entry created by the original store miss can be written in to the DLFB before it reloads the data into the data cache. This behavior is known as store miss merging.

12.6.1.5 Store Hit to a Data Cache Block Marked Shared

When a write-back store hits in the L1 data cache and the block is in the shared state, the target block is invalidated in the data cache. The store is then treated as a miss.

12.6.1.6 Data Cache Block Push Operation

When an L1 cache block in the core complex is snooped (by another bus master) and the data hits and is modified, the cache block must be written to memory and made available to the snooping device. The push operation propagates to the DWB and then to the CCB.

12.6.2 L1 Cache Block Replacement Selection

Both the instruction and data cache use the pseudo least-recently-used (PLRU) replacement algorithm described in this section when a new block needs to be placed in the cache. Note that data cache replacement selection is performed at reload time and not when the miss occurs. Instruction cache replacement selection occurs when an instruction cache miss is first recognized.

Each time a cache line is accessed, it is tagged as the most recently used (MRU) line of the set. When a miss occurs, if all lines in the set are valid (occupied), then the least recently used line is replaced with the new data. The PLRU bits in the cache are updated each time a cache hit occurs based on the most recently used cache line.

Modified data to be replaced is written into a DWB and is eventually written back to main memory.

Data load or write-back store accesses that miss in the L1 data cache function similarly to L1 instruction cache misses. They cause a new line to be allocated on a PLRU basis, provided the cache is not completely locked or disabled.

Note that modified data in the replacement line of any cache can cause a castout to occur to the CCB. In all of these cases, the castout is not initiated until the new data is ready to be loaded.

12.6.2.1 PLRU Replacement

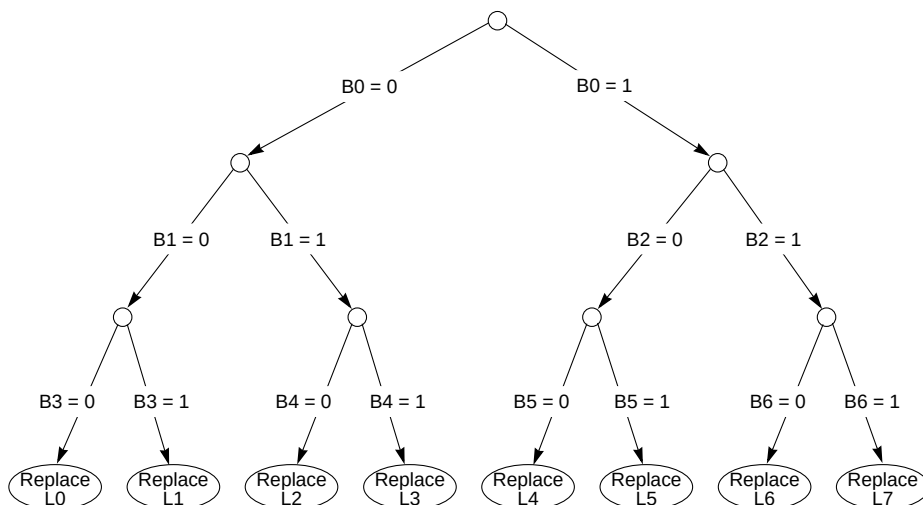
Block replacement is performed using a binary decision tree pseudo least-recently-used (PLRU) algorithm. There is an identifying bit for each way in the cache, L[0-7]. There are seven PLRU bits, B[0-6] for each set in the cache to determine the replacement victim (the line that needs to be cast out). The PLRU bits are updated every time there is a hit in the set or every time a new line is allocated or replaced.

This algorithm prioritize replacing invalid entries over valid ones (starting with way 0). Otherwise, if all ways are valid, a way is selected for replacement according to the PLRU bit encodings shown in Table 12-9.

Table 12-9. L1 PLRU Replacement Way Selection

If the PLRU bits are						Then the way selected for replacement is
B0	0	B1	0	B3	0	L0
	0		0		1	L1
	0		1	B4	0	L2
	0		1		1	L3
	1	B2	0	B5	0	L4
	1		0		1	L5
	1		1	B6	0	L6
	1		1		1	L7

Figure 12-4 shows the binary decision tree used to generate the victim line in the PLRU algorithm.

**Figure 12-4. PLRU Replacement Algorithm**

During power-up or hard reset, the valid bits of the L1 caches are automatically cleared to point to way L0 of each set.

12.6.2.2 PLRU Bit Updates

Except for snoop accesses, each time a cache block is accessed, it is tagged as the most recently used (MRU) way of the set. For every hit in the cache or when a new block is reloaded, the PLRU bits for the set are updated using the rules specified in Table 12-10.

Table 12-10. PLRU Bit Update Rules

If the current access is to	Then the PLRU bits in the set are changed to the following ¹						
	B0	B1	B2	B3	B4	B5	B6
L0	1	1	x	1	x	x	x
L1	1	1	x	0	x	x	x
L2	1	0	x	x	1	x	x
L3	1	0	x	x	0	x	x
L4	0	x	1	x	x	1	x
L5	0	x	1	x	x	0	x
L6	0	x	0	x	x	x	1
L7	0	x	0	x	x	x	0

¹ x = Does not change

Note that only three PLRU bits are updated for any access.

12.6.2.3 Cache Locking and PLRU

The core complex does not replace locked lines. Locking of cache lines in the L1 cache is accomplished through the lock bits. The lock bits are set through the cache locking instructions and are cleared through the cache unlocking instructions or line invalidation. The lock bits are used at reload time to steer the PLRU algorithm away from selecting cache lines that are locked.

Chapter 13

Memory Management Units

This chapter describes the implementation details of the e500 core complex MMU relative to the Book E architecture and the Motorola Book E standards. For background on the MMU definition in Book E and the Motorola Book E standards, see Chapter 11, “Cache and MMU Background.”

13.1 e500 MMU Overview

The e500 core complex employs a two-level memory management unit (MMU) architecture. There are separate data and instruction level 1 (L1) MMUs in hardware backed up by a unified level 2 (L2) MMU. The L1 MMUs are completely invisible with respect to the architecture. The programming model for implementing translation lookaside buffers (TLBs) provided in Book E and the Motorola Book E standard applies to the L2 MMU of the core complex.

13.1.1 MMU Features

The e500 core has the following features:

- 32-bit effective address translated to 32-bit real address (using a 41-bit interim virtual address)
- Two-level MMU containing a total of six TLBs for maximizing TLB hit rates
- Three 8-bit PID registers for supporting up to 255 translation IDs at any time in the TLB, with three concurrent translation IDs as potential matches for each access
- TLB entries for variable (4-Kbyte–256-Mbyte) and fixed-size (4-Kbyte) pages
- No page table format defined; software is free to use its own page table format.
- TLBs maintained by system software through the TLB instructions and six MAS registers

The level 1 MMUs have the following features:

- Two 4-entry, fully-associative TLB arrays (one for instruction accesses and one for data accesses) supporting the nine page sizes shown in Table 13-2.

- Two 64-entry, 4-way set-associative TLB arrays (one for instruction accesses and one for data accesses) that support only 4-Kbyte pages.
- L1 MMU access occurs in parallel with L1 cache access time (address translation/L1 cache access can be fully pipelined so one ld/st can be completed on every clock).
- Performs an L1 TLB lookup for an instruction access in parallel with an L1 TLB lookup for a data access.
- All L1 TLB entries are a proper subset of TLB entries resident in L2 MMU (completely maintained by the hardware).
- Automatically performs invalidations to maintain consistency with L2 TLBs.

The level 2 MMU has the following features:

- A 16-entry, fully-associative unified (for instruction and data accesses) L2 TLB array (TLB1) supports the nine page sizes shown in Table 13-2
- A 256-entry, 2-way set-associative unified (for instruction and data accesses) L2 TLB array (TLB0) supports only 4-Kbyte pages
- Hardware assistance for TLB miss exceptions
- TLB1 and TLB0 managed by **tlbre**, **tlbwe**, **tlbsx**, **tlbsync**, **tlbivax**, and **mtspr** instructions
- Performs invalidations in TLB1 and TLB0 caused by **tlbivax** instructions executed by this core. Also supports snooping of TLB1 and TLB0 for invalidation caused by **tlbivax** instructions executed by other masters.
- IPROT bit implemented in TLB1 prevents invalidations (positively affecting TLB hit rate)

13.1.2 TLB Entry Maintenance Features

The TLB entries of the e500 core complex must be loaded and maintained by the system software; this includes performing any required table search operations in memory. The e500 provides support for maintaining TLB entries in software with the resources shown in Table 13-1. Note that many of these features are defined at the Motorola Book E level.

Table 13-1. TLB Maintenance Programming Model

Features		Description	More Information Section/Page
TLB Instructions	tlbre	TLB Read Entry instruction	13.4.1/13-15
	tlbwe	TLB Write Entry instruction	13.4.2/13-16
	tlbsx rA, rB (preferred form is tlbsx 0, rB)	TLB Search for entry instruction	13.4.3/13-17
	tlbivax rA, rB	TLB Invalidate entries instruction	13.4.4/13-17
	tlbsync	TLB Synchronize invalidations with other masters' instruction	13.4.5/13-19
Registers	PID0–PID2	Process ID registers	See Table 13-7 for more comprehensive cross references
	MMUCSR0	MMU control and status register	
	MMUCFG	MMU configuration register	
	TLB0CFG–TLB1CFG	TLB configuration registers	
	MAS0–MAS4, MAS6	MMU assist registers. Note that MAS5 is not implemented on the e500.	
	DEAR	Data exception address register	
Interrupts	Instruction TLB miss exception	Causes instruction TLB error interrupt	13.5.1/13-21
	Data TLB miss exception	Causes data TLB error interrupt	13.5.2.2/13-22
	Instruction permissions violation exception	Causes ISI interrupt	
	Data permission violation exception	Causes DSI interrupt	

Other hardware assistance features for maintenance of the TLBs on the e500 are described in Section 13.5, “TLB Entry Maintenance—Details.”

13.2 Effective-to-Real Address Translation

The core complex fetch and load/store units generate 32-bit effective addresses. The MMU translates each of these addresses to a 32-bit real address which is then used for memory bus accesses. Figure 13-1 illustrates the high-level translation flow, showing that because the smallest page size supported by the e500 core complex is 4 Kbytes, the least significant 12 bits always index within the page and are untranslated. The appropriate L1 MMU (instruction or data) is checked for a matching address translation first. If it misses, the request for translation is forwarded to the unified (instruction and data) L2 MMU.

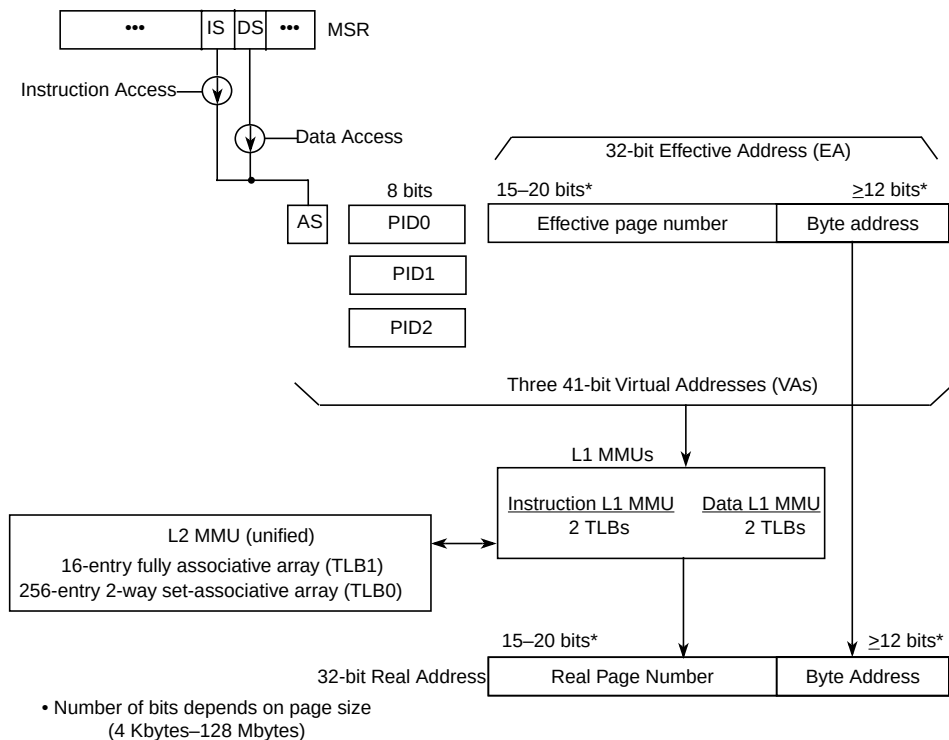


Figure 13-1. Effective-to-Real Address Translation Flow

13.2.1 Virtual Addresses with Three PID Registers

As shown in Figure 13-1, the address translation process starts with an effective address that is prepended with an address space (AS) value and a process ID to construct a virtual address (VA). A virtual address is then translated into a real address based on the translation information found in the on-chip TLB of the appropriate L1 MMU. The address space (AS) bit for the access is selected from the value of MSR[IS] or MSR[DS], for instruction or data accesses, respectively.

The e500 constructs three virtual addresses for each access. The core complex implements three process ID (PID) registers, PID0–PID2, as SPRs shown in Section 2.12.1, “Process ID Registers (PID0–PID2).” All of the current values in the PID registers are used in the TLB look-up process and are compared with the TID field in all the TLBs. If any of the PID values in PID0–PID2 matches with a TLB entry in which all the other match criteria are met, that entry is used for translation.

Note that when a TID value in a TLB entry is all zeros, it causes a “match” in the PID compare (effectively ignoring the values of the PID registers). Thus, the operating system

can set the values of all the TIDs to zero, effectively eliminating the PID values from all translation comparisons.

13.2.2 Variable-Sized Pages

There are two kinds of TLBs on the e500 core complex as follows:

- TLBs that translate addresses for 4-Kbyte pages only. These TLBs are set-associative based on the page number (page address).
- TLBs that translate addresses for variable-sized pages. These TLBs are fully-associative.

Table 13-2 shows the nine page sizes supported by the fully-associative TLBs that support variable-sized pages (VSPs) on the e500 core complex.

Table 13-2. Page Sizes for L1VSPs and TLB1 (L2 MMU) on the e500 Core

Page Sizes
4 Kbyte
16 Kbyte
64 Kbyte
256 Kbyte
1 Mbyte
4 Mbyte
16 Mbyte
64 Mbyte
256 Mbyte

For more information on the bit ranges of effective page numbers and offsets that are translated for these pages sizes, see Section 11.2.3.1, “Page Size and Effective Address Bits Compared,” and Section 11.2.3.3, “Page Size and Real Address Generation.”

13.2.3 Checking for TLB Entry Hit

Figure 13-2 shows the compare function used by the e500 to check the MMU structures for a hit for the three virtual addresses that correspond to the instruction or data access (one virtual address for each current PID register value). Note that this figure is functionally similar to the figure in Section 11.2.3, “Address Translation,” that shows the Book E algorithm, except that this figure shows that three PID values are compared for each access.

A hit to multiple matching TLB entries is considered a programming error. If this occurs, the TLB generates an invalid address and TLB entries may be corrupted (an exception is not reported).

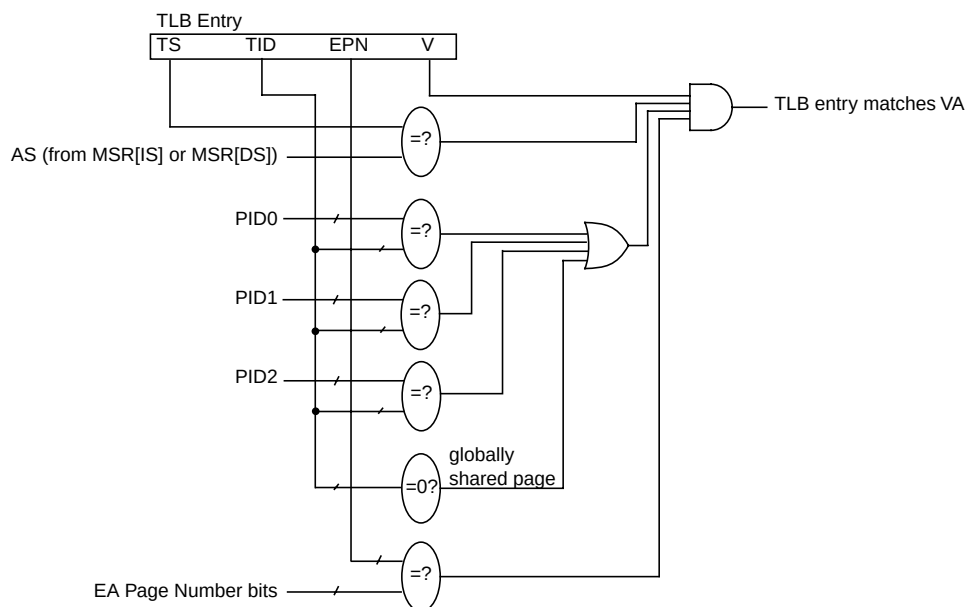


Figure 13-2. Virtual Address and TLB-Entry Compare Process

13.2.4 Checking for Access Permissions

When a TLB entry matches with one of the three virtual addresses of an access, the permission bits of the TLB entry are compared with attribute information of the access (read/write, instruction/data, user/supervisor) in order to see if the access is allowed to that page. The checking of permissions on the e500 functions is described in Section 11.2.4, “Permission Attributes.”

13.3 Translation Lookaside Buffers (TLBs)

The e500 core complex implements six TLB arrays to maximize address translation performance and to provide ample flexibility for the operating system. Figure 13-3 contains a more detailed description of the 2-level MMU structure. Note that for an instruction access, both the I-L1VSP and the I-L1TLB4K are checked in parallel for a TLB hit. Similarly, for a data access, both the D-L1VSP and the D-L1TLB4K are checked in parallel for a TLB hit. The instruction L1 MMU and data L1 MMU operate independently and can be accessed in parallel, so that hits for instruction accesses and data accesses can occur in the same clock.

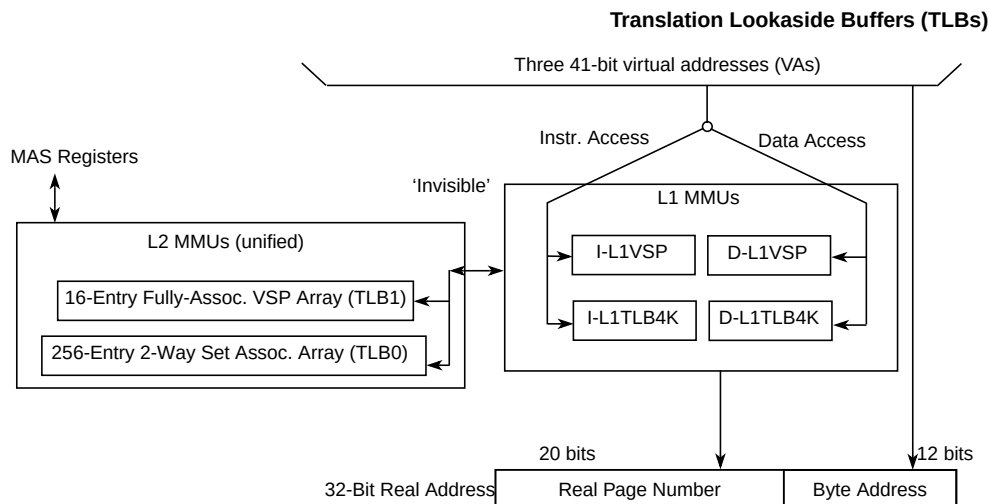


Figure 13-3. Two-Level MMU Structure

Additionally, Figure 13-3 shows that when the L2 MMU is checked for a TLB entry, both TLB1 and TLB0 are checked in parallel. It also identifies the L1 MMUs as invisible to the programming model (not accessible to the operating system); they are managed completely by the hardware. Conversely, the L2 MMU is accessed by the TLB instructions by way of the MAS registers.

A hit to multiple TLB entries in the L1 MMU (even if they are in separate arrays) is considered to be a programming error. This is also the case if an access results in a hit to multiple TLB entries in the L2 MMU. If this occurs, the TLB generates an invalid address and TLB entries may be corrupted (an exception is not reported).

Table 13-3 lists the various TLBs and describes their characteristics.

Table 13-3. Index of TLBs

Location	Name	Page Sizes Supported	Associativity	Size of TLB (# of entries)	Instruction/Data Translations	Filled by
Instruction L1 MMU	I-L1VSP	9 page sizes ¹	Fully assoc.	4	instruction	TLB1 hit
	I-L1TLB4K	4 Kbyte	4-way	64	instruction	TLB0 hit
Data L1 MMU	D-L1VSP	9 page sizes ¹	Fully assoc.	4	data	TLB1 hit
	D-L1TLB4K	4 Kbyte	4-way	64	data	TLB0 hit
L2 MMU	TLB1	9 page sizes ¹	Fully assoc.	16	unified (I and D)	tlbwe instruction
	TLB0	4 Kbyte	2-way	256	unified (I and D)	tlbwe instruction

¹ See Table 13-2 for supported page sizes.

13.3.1 L1 TLB Arrays

As shown in Figure 13-1, there are two level 1 (L1) MMUs in the core complex. As shown in Figure 13-3 and Table 13-3, the instruction and data L1 MMUs each implement a 4-entry, fully associative L1VSP array and a 64-entry, 4-way set associative L1TLB4K array, comprising the following L1 MMU arrays:

- Instruction L1VSP—4-entry, fully-associative
- Instruction L1TLB4K—64-entry, 4-way set-associative
- Data L1VSP—4-entry, fully associative
- Data L1TLB4K—64-entry, 4-way set associative

As their names imply, the L1TLB4K arrays only support a 4-Kbyte page size while the L1VSP arrays support nine page sizes. To perform a lookup for instruction accesses, both the L1TLB4K and the L1VSP TLBs in the instruction MMU are searched in parallel for the matching TLB entry. Similarly, for data accesses, both the L1TLB4K and the L1VSP TLBs in the data MMU are searched in parallel for the matching TLB entry. The contents of a matching TLB entry are then concatenated with the page offset of the original effective address; the bit range that is translated is determined by the page size. The result constitutes the real (physical) address for the access.

Figure 13-4 shows the organization of the L1 TLBs in both the instruction and data L1 MMUs.

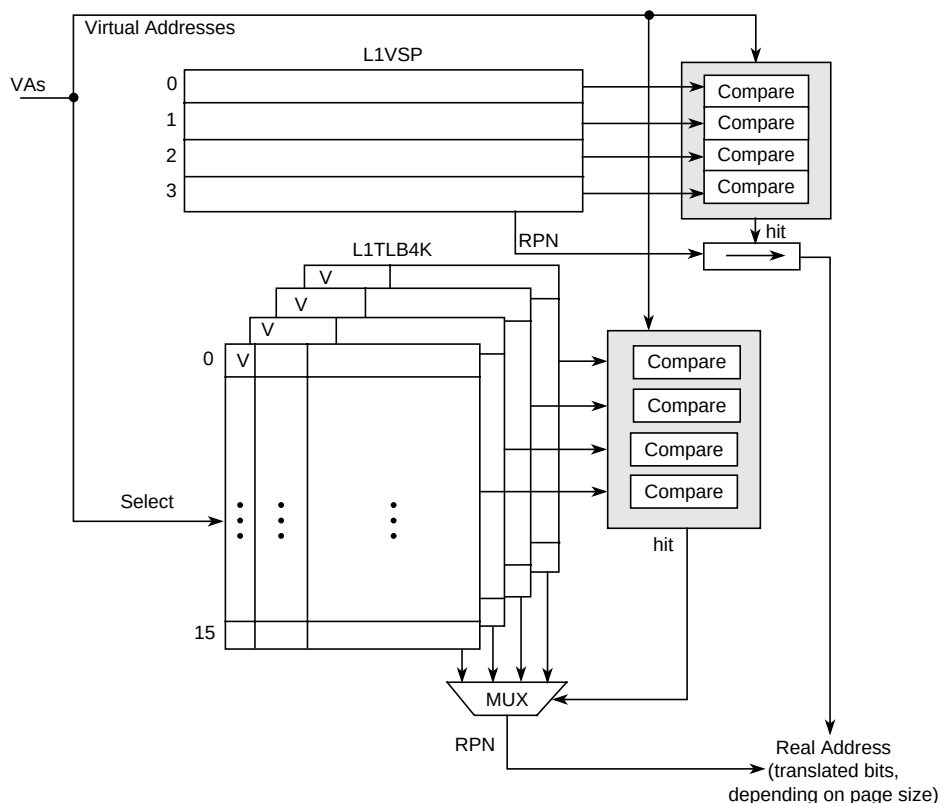


Figure 13-4. L1 MMU TLB Organization

L1TLB4K TLB entries are replaced based on a true LRU algorithm. The L1VSP entries are also replaced based on a true LRU replacement algorithm. The LRU bits are updated each time a TLB entry is accessed for translation. However, there are other speculative accesses performed to the L1 MMUs that cause the LRU bits to be updated. The performance of the L1 MMUs is high, even though it is not possible to predict (externally) exactly which entry is the next to be replaced.

13.3.2 L2 TLB Arrays

The level 1 MMUs are backed up by a unified level 2 MMU that translates both instruction and data addresses. Like each L1 MMU, the L2 MMU consists of two TLB arrays:

- TLB1: a 16-entry, fully associative array that supports nine page sizes.
- TLB0: a 256-entry, 2-way set associative array that supports only 4-Kbyte page sizes.

Translation Lookaside Buffers (TLBs)

The two L2 TLBs, which are the only TLBs accessible to the software, are shown in Figure 13-5.

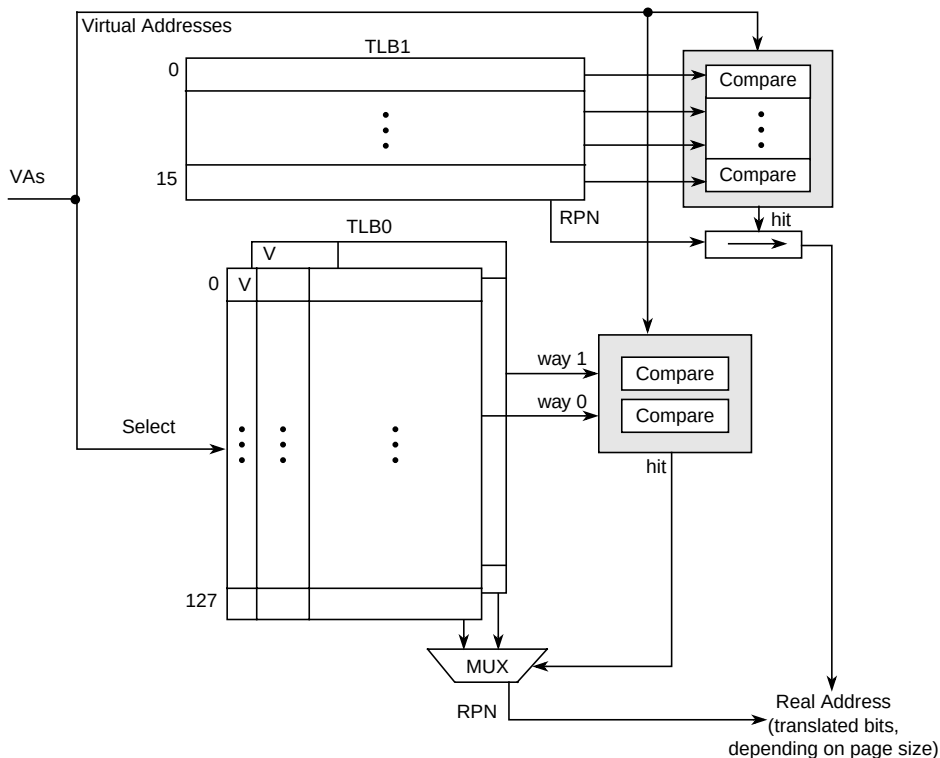


Figure 13-5. L2 MMU TLB Organization

13.3.2.1 IPROT Invalidation Protection in TLB1

The IPROT bit in TLB1 is used to protect TLB entries from invalidation. TLB1 entries with IPROT set can never be invalidated by a **tlbivax** instruction executed by this processor (internal), by an external **tlbivax** instruction, or by a flash invalidate initiated by writing to the MMUCSR0. The IPROT bit can be used to protect critical code and data such as interrupt vectors/handlers in order to guarantee that the instruction fetch of those vectors never takes a TLB miss exception. Entries with IPROT set can only be invalidated by writing a 0b0 to the valid bit of the entry (by using the MAS registers and executing the **tlbwe** instruction).

Only TLB entries in TLB1 can be protected from invalidation; entries in TLB0 and in the level 1 MMUs can not be protected from invalidation (they don't implement the IPROT bit). See Section 11.3.4, "TLB Entry Invalidations," for more background information on the IPROT attribute.

Invalidation operations are guaranteed to invalidate the entry that translates the address specified in the operand of the **tlbivax** instruction. Other entries may also be invalidated by this operation if they are not protected with IPROT. A precise invalidation can be performed by writing a 0b0 to the valid bit of a TLB entry. Note that successful invalidation operations in the L2 MMU also invalidate matching entries in the L1 MMU.

If HID1[ABE] = 1, enabling broadcast operations on the core complex bus (CCB), execution of **tlbivax** is broadcast onto the CCB, regardless of whether or not the invalidation was successful. Flash invalidations (initiated by writing to the appropriate bits in MMUCSR0) are never broadcast.

13.3.2.2 SHAREN Page Attribute

The core complex defines the SHAREN bit in the TLBs that designates pages as shared or not, with respect to the memory/cache subsystem. The value of the SHAREN page attribute designates the use of either the MEI or MESI cache coherency protocol for that page. If SHAREN = 1, loads (and cache accesses treated as loads) that miss in the L1 cache go out on the CCB as read accesses instead of read-with-intent-to-modify (RWITM) accesses. Note that the value of SHAREN does not affect snooping for that page.

13.3.2.3 Replacement Algorithms for L2 MMU

The replacement algorithm for TLB1 (the fully associative TLB in the L2 MMU) must be implemented completely by the system software. Thus, when an entry in TLB1 is to be replaced, the software selects which entry to replace and writes the entry number to the MAS0[ESEL] field before executing a **tlbwe** instruction.

TLB0 entry replacement is also implemented by software. To assist the software with TLB0 replacement, the e500 core complex provides a hint that can be used for implementing a round-robin replacement algorithm. Because TLB0 is two-way set associative, the only parameter required to select the entry to replace is the way select value for the new entry: zero or one. (The entry within the way is selected by EA[45–51].) The mechanism for the round-robin replacement uses the following bits as shown in Figure 13-6:

- TLB0[NV]—the next victim bit within TLB0,
- MAS0[NV]—the next victim bit of MAS0, and
- MAS0[ESEL]—selects the way to be replaced on **tlbwe**

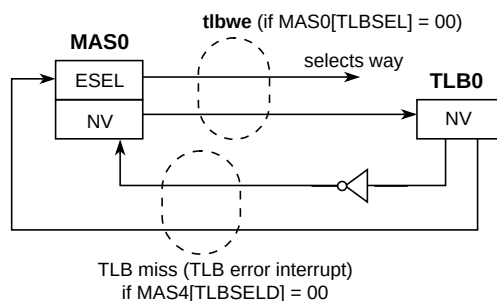


Figure 13-6. Round Robin Replacement for TLB0

On execution of a **tlbwe** instruction, MAS0[ESEL] selects the way of TLB0 to be loaded (way 0 or way 1). Also, when MAS0[TLBSEL] = 00 (selecting TLB0), TLB0[NV] is loaded with the MAS0[NV] value on execution of a **tlbwe** instruction. In addition, when a TLB miss exception occurs (causing a TLB error interrupt), if MAS4[TLBSELD] = 00, the hardware automatically loads the current value of TLB0[NV] into MAS0[ESEL] and the complement of TLB0[NV] into MAS0[NV]. This sets up MAS0 such that if those values are not overwritten, the alternate way will be selected on the next execution of a **tlbwe** instruction, effectively alternating between way 0 and way 1 for writing TLB0 entries.

See Table 13-14 for a complete description of MAS register updates on various exception conditions.

Note that the system software can load any value into MAS0[ESEL] and MAS0[NV] prior to execution of **tlbwe**, effectively overwriting this round robin replacement algorithm. In this case, the value written by software into MAS0[NV] is used as the next TLB0[NV] value on a TLB miss.

13.3.3 Consistency Between L1 and L2 TLBs

The contents of the L1 TLBs are always a proper subset of the TLB entries currently resident in the L2 MMU. They serve to improve performance because they have a faster access time than the larger L2 TLBs. The relationships between the six TLBs are shown in Figure 13-7.

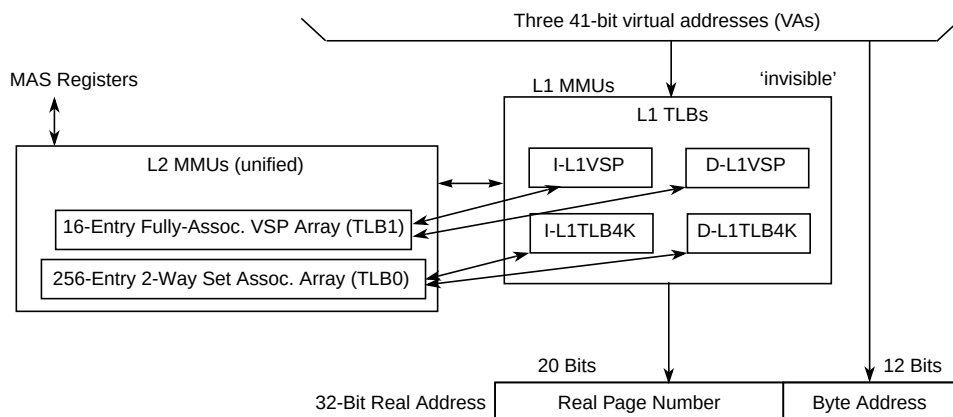


Figure 13-7. L1 MMU TLB Relationships with L2 TLBs

On an L1 MMU miss, L1 MMU array entries are automatically reloaded using entries from their level 2 array equivalent. For example, if the L1 data MMU misses but there is a hit for one of the three virtual addresses in TLB1 (the 16-entry fully-associative L2 TLB), the matching entry is automatically loaded into the data L1VSP array. Likewise, if the L1 data MMU misses, but there is a hit for the access in TLB0 (the 64-entry TLB for 4-Kbyte pages), the matching entry is automatically loaded into the data L1TLB4K array.

A hit for a single access to multiple TLB entries in the L2 MMU (even if they are in separate arrays) is considered to be a programming error. If this occurs, the TLB generates an invalid address and TLB entries may be corrupted (an exception is not reported).

Note that when an L2 TLB entry is invalidated by executing a **tlbwe** instruction that clears a valid bit, any corresponding entry in the L1 TLB arrays is also automatically invalidated. Plus, when L2 TLB entries are invalidated by the execution of **tlbivax**, by the detection of a TLB invalidate command broadcast by another processor, or by a flash invalidate operation, corresponding L1 TLB entries are also invalidated as described in Section 13.4.4, “TLB Invalidate (**tlbivax**) Instruction.”

13.3.4 L1 and L2 TLB Access Times

The L1 TLB arrays are checked for a translation hit in parallel with the on-chip L1 cache lookups and incur no penalty on an L1 TLB hit. If the L1 TLB arrays miss, the access proceeds to the L2 TLB arrays. For L1 instruction address translation misses, the L2 TLB latency is at least 5 clocks; for L1 data address translation misses, the L2 TLB latency is at least 6 clocks. These access times may be longer depending on some arbitration performed by the L2 arrays for simultaneous instruction L1 TLB misses, data L1 TLB misses, the execution of TLB instructions, and TLB snoop operations (snooping of TLBINV operations on the CCB).

Note that when a TLBINV operation is detected on the CCB, the L2 MMU arrays become inaccessible due to the snooping activity caused by the TLBINV.

13.3.5 The G Bit (of WIMGE)

The G bit provides protection from bus accesses due to speculative and faultable instruction execution. A speculative access is defined as an access caused by an instruction that is downstream from an unresolved branch. A faultable access is defined as an access that could be cancelled due to an exception on an uncompleted instruction.

On the e500, if the page for this type of access is marked with G = 1 (guarded), the access stalls (if it misses in the cache) until the exception status of any instructions in progress is known. If G = 0 (unguarded), this type of access is issued to the CCB regardless of the completion status of other instructions.

Note that after an access with G = 1 is begun to the CCB, it is guaranteed to be completed. That is, after the address tenure is acknowledged on the CCB, the core completes the access, even if an asynchronous interrupt is pending.

The G bit is ignored for instruction fetches, and instructions are speculatively fetched from guarded pages. To prevent speculative fetches from pages that do not contain instructions and are guarded, the page should be also designated as no-execute (with the UX/SX page permission bits cleared).

13.3.6 TLB Entry Field Definitions

Table 13-4 summarizes the fields of e500 TLB entries. Note that all of these fields are defined at the Motorola Book E level except for SHAREN, which is an implementation-dependent feature of the core complex. See Section 11.3.2, “TLB Entries,” for the definition of TLB fields at the Motorola Book E level.

Table 13-4. TLB Entry Bit Definitions

Field	Comments
V	Valid bit for entry
TS	Translation address space (compared with AS bit of the current access)
TID[0–7]	Translation ID (compared with PID0, PID1, PID2 or TIDZ (all zeros))
EPN[0–19]	Effective page number (compared with EA[31–50] for 4 Kbyte pages)
RPN[0–19]	Real page number (translated address RA[31–50] for 4 Kbyte pages)

Table 13-4. TLB Entry Bit Definitions

Field	Comments
SIZE[0–3]	Encoded page size 0000 Reserved 0001 4 Kbyte 0010 16 Kbyte 0011 64 Kbyte 0100 256 Kbyte 0101 1 Mbyte 0110 4 Mbyte 0111 16 Mbyte 1000 64 Mbyte 1001 256 Mbyte all others—reserved
PERMIS[0–5]	Supervisor execute, write, and read permission bits, and user execute, write, and read permission bits.
WIMGE	Memory/cache attributes (write-through, cache-inhibit, memory coherence required, guarded, endian)
SHAREN	Cache sharing state for L1 caches 0 Fills use the exclusive memory state (read transaction on the CCB) 1 Fills use the shared memory state (RWITM transaction on the CCB)
X0, X1	Extra system attribute bits (for definition by system software)
U0–U3	User bits—used only by software. These bits exist in the L2 MMU TLBs only (TLB1 and TLB0)
IPROT	Invalidation protection (exists in TLB1 only)

13.4 TLB Instructions—Implementation

The implementation of the **tlbre**, **tlbwe**, **tlbsx**, **tlbivax**, and **tlbsync** instructions is summarized in this section. The extended (64-bit) forms of these instructions are invalid for the core complex. See Section 3.2, “Unsupported Instructions and Instruction Forms.” Although the **tlbre**, **tlbwe**, **tlbsx**, **tlbivax**, and **tlbsync** instructions are defined by Book E, their specific functions are defined by Motorola Book E.

13.4.1 TLB Read Entry (tlbre) Instruction

The **tlbre** instruction causes the contents of a single TLB entry to be extracted from the L2 MMU and placed in the corresponding fields of the MMU assist (MAS) registers. The entry extracted is specified by the TLBSEL, ESEL, and EPN fields of the MAS0, and MAS2 registers. The contents extracted from the L2 MMU are placed in the MAS0, MAS1, MAS2, and MAS3 registers. See Section 13.7.2, “MAS Register Updates,” for details on which MAS register fields are updated.

The following RTL describes the e500 core complex **tlbre** implementation:

```

tlb_entry_id = MAS0(TLBSEL, ESEL) || MAS2(EPN)
result = L2MMU(tlb_entry_id)
MAS0, MAS1, MAS2, MAS3 = result

```

Note that if the instruction specifies a TLB entry that is not found, the results placed in MAS0–MAS3 are undefined. See Section 11.3.3.1, “Reading TLB Entries,” for information at the Motorola Book E level.

13.4.1.1 Reading Entries from the TLB1 Array

Entries in TLB1 can be read by first writing the necessary entry-identifying information into MAS0 using **mtspr** and then executing the **tlbre** instruction. To read an entry from TLB1, MAS0[TLBSEL] must be = 01 and MAS0[ESEL] must be set to point to the desired entry. After executing the **tlbre** instruction, MAS0–MAS3 are updated with the data from the selected TLB entry in TLB1.

13.4.1.2 Reading Entries from the TLB0 Array

Entries in TLB0 can be read by first writing the necessary entry-identifying information into MAS0 and MAS2 using **mtspr** and then executing the **tlbre** instruction. To read an entry from TLB0, MAS0[TLBSEL] must be = 00, MAS0[ESEL] must be set to point to the desired way, and EPN[45–51] in MAS2 must be loaded with the desired index. After executing the **tlbre** instruction, MAS0–MAS3 are updated with the data from the selected TLB entry in TLB0.

13.4.2 TLB Write Entry (tlbwe) Instruction

The **tlbwe** instruction causes the contents of certain fields of the MAS registers (MAS0, MAS1, MAS2, and MAS3) to be written into a single TLB entry in the L2 MMU. The entry written is specified by the TLBSEL, ESEL, and EPN fields of the MAS0, and MAS2 registers.

The following RTL describes the e500 core complex **tlbwe** implementation:

```
tlb_entry_id = MAS0(TLBSEL, ESEL) || MAS2(EPN)
L2MMU(tlb_entry_id) = MAS0, MAS1, MAS2, MAS3
```

Note that when an L2 TLB entry is written, it may be displacing an already valid entry in the same L2 TLB location (a victim). If a valid L1 TLB entry corresponds to the L2 MMU victim entry, that L1 TLB entry is automatically invalidated. See Section 11.3.3.2, “Writing TLB Entries,” for synchronization requirements defined at the Motorola Book E level for the use of **tlbwe**.

13.4.2.1 Writing to the TLB1 Array

TLB1 can be written by first writing the necessary information into MAS0–MAS3 using **mtspr** and then executing the **tlbwe** instruction. To write an entry into TLB1, MAS0[TLBSEL] must = 01, and MAS0[ESEL] must point to the desired entry. When the **tlbwe** instruction is executed, the TLB entry information stored in MAS1–MAS3 is written into the selected TLB entry in the TLB1 array.

13.4.2.2 Writing to the TLB0 Array

TLB0 can be written by first writing the necessary information into MAS0–MAS3 using **mtspr** and then executing the **tlbwe** instruction. To write an entry into TLB0, MAS0[TLBSEL] must = 00, MAS0[ESEL] must point to the desired way, and EPN[45–51] in MAS2 must be loaded with the desired index. When the **tlbwe** instruction is executed, the TLB entry information stored in MAS1–MAS3 is written into the selected TLB entry in TLB0.

13.4.3 TLB Search (tlbsx) Instruction—Searching the TLB1 and TLB0 Arrays

The **tlbsx** instruction updates the MAS registers conditionally based on the success or failure of a TLB lookup in the L2 MMU. The lookup is controlled by the effective address provided by GPR[rA] + GPR[rB] specified in the instruction encoding, as well as by the SAS and SPID0 search fields in MAS6. The values placed into MAS0, MAS1, MAS2, and MAS3 differ, depending on whether a successful or unsuccessful search occurred. See Section 13.7.2, “MAS Register Updates,” for details on which MAS register fields are updated for these cases.

Note that **rA** = 0 is the preferred form for **tlbsx** and that some Motorola implementations, such as the e500, take an illegal instruction exception program interrupt if **rA**!=0.

The following RTL describes the e500 core complex **tlbsx** implementation:

```
if RA!=0 then generate illegal exception
EA = 320 || GPR(RB)32:63
ProcessID = MAS6(SPID0), 0b0000_0000
AS = MAS6(SAS)
VA = AS || ProcessID || EA
if Valid_TLB_matching_entry_exists (VA)
then result = see Table 13-14, column labelled "tlbsx hit"
else result = see Table 13-14, column labelled "tlbsx miss"
MAS0, MAS1, MAS2, MAS3 = result
```

The **tlbsx** instruction searches both the TLB1 and TLB0 arrays using EPN[32–51] from the GPR used as the instruction operand, and the SAS (search AS bit) and SPID0 (search PID) values from MAS6. If the search results in a hit, the information for the TLB entry that hit is loaded into MAS0–MAS3. The valid bit in MAS1 is used as the success flag as follows:

- If the search is successful, MAS1[V] is set
- If the search is unsuccessful, MAS1[V] is cleared.

The **tlbsx** instruction is especially useful for finding the TLB entry that caused a DSI or ISI exception. In this case, at most four **tlbsx** instructions are required: one for each of the current PID values, and one with a PID value of 0x00.

13.4.4 TLB Invalidate (tlbivax) Instruction

The following RTL describes the e500 core complex **tlbivax** implementation:

```

if RA = 0, a = 0
else, a = GPR(RA)
EA = a + GPR(RB)
if valid_TLB_matching_entry exists
then invalidate

```

A TLB invalidate operation is performed whenever a **tlbivax** instruction is executed. This instruction invalidates any TLB entry that corresponds to the virtual addresses calculated by this instruction. This operation includes invalidating TLB entries contained in TLBs on other processors and devices in addition to the processor executing the **tlbivax** instruction. Thus an invalidate operation is broadcast throughout the coherent domain of the processor executing this instruction.

Because the virtual address can be much larger than the physical address, the full virtual address specified by the **tlbivax** instruction cannot be broadcast to all devices. Instead, a subset address is broadcast that fits within the space of the implemented physical addressing model.

The address that is broadcast by the core complex is detailed in Table 13-5. Note that this subset address is both broadcast to other processors and is used by the processor executing the **tlbivax** instruction. Thus, no other information except for that shown in Table 13-5 is used for the invalidation. As shown in the table, the bits of effective address used to perform the **tlbivax** invalidation of TLB1, TLB0, and the L1 TLBs are bits 32–51 of $rA + rB$.

Table 13-5. tlbivax EA Bit Definitions

Bits of ($rA + rB$) (preferred form is for $rA = 0$)	Meaning	More Information Section/Page
32–51	EA[32–51] for invalidation matching	—
52–59	Reserved; should be zero	—
60	TLBSEL. Selects which TLB is targeted for invalidation 0 TLB0 1 TLB1	13.4.4.1/13-19
61	INV_ALL command	13.4.4.2/13-19
62–63	Reserved	—

The limited virtual address used to invalidate TLB entries has the side effect that a single **tlbivax** instruction can invalidate more than a single entry in a targeted TLB. This is because the **tlbivax** does not compare the values of the PID or AS bits. A **tlbivax** targeted at TLB0 can invalidate either or both ways within an TLB0 index and up to all four ways within an L1TLB4K index. Also, a **tlbivax** targeted at TLB1 can invalidate up to all 16 entries in the array, or up to all 8 entries of the L1VSPs (instruction and data).

The **tlbivax** instruction invalidates all matching entries in the instruction and data L1 TLBs simultaneously. Also, the core complex always snoops TLB invalidate transactions from other CCB bus masters (if any) and invalidates matching TLB entries accordingly.

Note that entries in TLB1 can be protected from invalidation by the **tlbivax** instruction by setting the IPROT bit for those entries. See Section 13.3.2.1, “IPROT Invalidation Protection in TLB1,” for more information. Also, refer to Section 11.3.4, “TLB Entry Invalidations,” for more information on the use of the IPROT bit defined for Motorola Book E processors.

13.4.4.1 TLB Selection for **tlbivax** Instruction

Because only a limited subset of the virtual address can be broadcast, extra information about the targeted TLB entries is encoded in two of the lower bits of the effective address calculated by the **tlbivax** instruction. Bit 60 of the **tlbivax** effective address is interpreted as the TLBSEL field. This bit indicates whether TLB1 or TLB0 is targeted by the invalidate operation. Because only a few bits (32–51) of address are broadcast and can be used in the invalidate comparison for TLB1, and most of those bits are masked out for larger page sizes, the TLBSEL field avoids unnecessary invalidations of large superpages in TLB1 when the **tlbivax** is targeting TLB0.

13.4.4.2 Invalidate All Address Encoding for **tlbivax** Instruction

Bit 61 of the **tlbivax** effective address is interpreted as the INV_ALL command. If this bit is set, it indicates that the invalidate operation should completely invalidate all entries of either TLB1 or TLB0 as indicated by the TLBSEL field, and invalidate all corresponding L1 TLB entries. Note that entries in TLB1 can be protected from this type of invalidation by setting the IPROT bit as described in Section 13.3.2.1, “IPROT Invalidation Protection in TLB1.”

13.4.4.3 TLB Invalidate Broadcast Enabling

In addition to invalidating the local matching TLB entries, the **tlbivax** instruction operation is also broadcast on the bus (causing a TLBINV address-only transaction) according to the value of the ABE (address broadcast enable) bit in the HID1 register as follows:

- If HID1[ABE] = 0, **tlbivax** instructions are not broadcast.
- If HID1[ABE] = 1, **tlbivax** instructions are broadcast.

13.4.5 TLB Synchronize (**tlbsync**) Instruction

The **tlbsync** instruction causes a TLBSYNC transaction on the CCB. This transaction is retried if any processor, including the one that executed the **tlbsync** instruction, has pending memory accesses that were issued before any previous **tlbivax** instructions were completed. This instruction effectively synchronizes the invalidation of TLB entries; **tlbsync** does not complete until all memory accesses caused by instructions issued before an earlier **tlbivax** instruction have completed.

13.5 TLB Entry Maintenance—Details

The TLB entries of the e500 core complex must be loaded and maintained by the system software, including performing the required table search operations in memory. However, the e500 provides some hardware assistance for these software tasks. Note that the system software cannot directly access the L1 TLBs, and the L1 TLBs are completely and automatically maintained in hardware as a subset of the contents of the L2 TLBs.

In addition to the resources described in Table 13-1, hardware assistance on the core complex for maintenance of TLB entries includes:

- Automatic loading of MAS0–2 based on the default values in MAS4 on TLB miss exceptions. This automatically generates most fields of the required TLB entry on a miss. Thus software should load MAS4 with likely values to be used in the event of a TLB miss condition.
- Automatic loading of the data exception address register (DEAR) with the effective address of the load, store, or cache management instruction that caused an alignment, data TLB miss (data TLB error interrupt), or permissions violation (DSI interrupt).
- Automatic loading into SRR0 of the effective address of the instruction that causes a TLB miss exception or a permissions violation.
- Automatic updates of the next victim (NV) field and MAS0[ESEL] fields for TLB0 entry replacement on TLB misses (TLB error interrupts); this occurs if TLBSELD = 00. See Section 13.3.2.3, “Replacement Algorithms for L2 MMU.”
- When **tlbwe** is executed, the information for the selected victim is read from the selected L2 TLB (TLB1 or TLB0). The victim’s EPN and TS are sent to both L1 MMUs to provide back-invalidation. Thus if the selected victim in the L2 MMU is also resident in an L1 MMU, it is invalidated (or victimized) in the L1 MMU. This forces inclusion in the TLB hierarchy. Additionally, the new TLB entry contained in MAS0–MAS3 is written into the selected TLB.

Note that while the **tlbwe** instruction loads an entry in the L2 TLB array, it does not load an entry in the L1 TLB array. The L1 arrays are loaded with new entries (automatically by the hardware) only when an access misses in the L1 array, but hits in a corresponding L2 array.

See Section 13.7.2, “MAS Register Updates,” for a complete description of automatic fields loaded into the MAS registers on execution of TLB instructions and for various exception conditions.

The following sections provide information on some of the actions taken by Motorola Book E devices on MMU exceptions:

- Section 11.3.6.1, “TLB Miss Exception Types”
— Section 11.3.6.1.1, “Instruction TLB Error Interrupt Settings”

- Section 11.3.6.1.2, “Data TLB Error Interrupt Settings”
- Section 11.3.6.1.3, “TLB Miss Exception MAS Register Settings”
- Section 11.3.6.2, “Permissions Violation Exception Types”
 - Section 11.3.6.2.1, “Instruction Storage Interrupt Settings”
 - Section 11.3.6.2.2, “Data Storage Interrupt Settings”

The following subsections provide supplementary information to the above references that applies for the e500.

13.5.1 Automatic Updates—TLB Miss Exceptions

When a TLB miss exception occurs, MAS0–MAS2 are automatically updated using the defaults specified in MAS4, as well as the AS and EPN[32–51] values corresponding to the access that caused the exception, as described in Section 13.7.2, “MAS Register Updates.”

In addition, if TLBSELD = 00 (selecting TLB0), MAS0[ESEL] is updated with the next victim information for TLB0. Finally, the MAS0[NV] bit is updated with the complement of TLB0[NV]. Thus, ESEL points to the current victim (the entry to be replaced), while MAS0[NV] points to the next victim to be used if a TLB0 entry is replaced. See Section 13.3.2.3, “Replacement Algorithms for L2 MMU,” for more information.

The process described above sets up all the TLB entry data necessary for a TLB write except for the RPN[32–51], the U0–U3 user bits, and the UX, SX, UW, SW, UR, and SR permission bits for the new entry, all of which are stored in MAS3. Thus, if the defaults stored in MAS4 are applicable to the TLB entry to be loaded, the TLB miss exception handler only has to update MAS3 with an **mtspr** before executing **tlbwe**. If the defaults are not applicable to the TLB entry being loaded, then the TLB miss exception handler must update MAS0–MAS2 appropriately before performing the TLB write. See Section 13.5.2, “TLB Interrupt Routines,” for more information on the handling of TLB miss exceptions.

13.5.2 TLB Interrupt Routines

When an exception is reported by the MMUs, the machine drains (that is, all instructions dispatched prior to the exception are executed). After all instructions are completed, the interrupt is acknowledged and MAS0–MAS3 are loaded as described in Section 13.5.1, “Automatic Updates—TLB Miss Exceptions.”

As is recommended for most interrupt handler routings, the TLB miss, DSI, and ISI exception handlers must first save the values of enough GPRs so that the handler has enough GPRs available for its own use. The handler should then perform an **mfcrr** to copy the CR data into one of the GPRs. Before exiting the handler, an **mtcrf** must be executed to restore the CR, and then the original GPR data must be restored.

The PID0–2 registers must also be restored (if modified) before exiting the handler. Note that PID register updates must be followed by an **isync**. This **isync** instruction must reside in an instruction page that is valid before the changes are made to the PID.

Some example code sequences showing exception routines (TLB miss, DSI) will be provided in future revisions of this document.

13.5.2.1 TLB Error Interrupt Handlers

TLB error interrupt handlers should also write the default PID value, as specified by MAS4[TIDSELD], into MAS6[SPID0]. If TIDSELD is 0b11 then 0x00 should be written to MAS6[SPID0] so that all entries will match in the PID field if a **tlbsx** instruction is executed. The handler is then free to modify the PID registers as needed to perform the necessary table search operation in software.

13.5.2.2 Permissions Violations (ISI, DSI) Interrupt Handlers

The only difference between the definition of actions on a permissions violation for Motorola Book E devices and for the e500 is that the e500 does not implement MAS5.

The permissions violations handlers can use the **tlbsx** instruction to load all necessary information about the faulting access into the MAS registers and make the appropriate changes. If the access was an instruction or data access, the handler can load the following effective address into **rB** in order to load the faulting TLB entry into the MAS registers:

- Instruction access: load SRR0 value into **rB**
- Data access: load DEAR value into **rB**

See Section 13.4.3, “TLB Search (**tlbsx**) Instruction—Searching the TLB1 and TLB0 Arrays,” for more information about the actions performed by the **tlbsx** instruction.

The guidelines for the saving and restoring of resources for permissions violations interrupt handlers are the same as that for TLB error interrupts except that the software in the DSI and ISI exception handlers need not modify MAS6[SPID0] (because it is automatically loaded with the value of PID0).

13.6 TLB States after Reset

During reset, all TLB entries in the L1 and L2 MMUs are flash invalidated. Then entry 0 of TLB1 is loaded with the values shown in Table 13-6. Note that because the core complex fetches from effective address 0xFFFF_FFFC out of reset, the first access out of reset is automatically translated with this default TLB entry. The instruction located at 0xFFFF_FFFC should be a branch instruction to the beginning of this 4-Kbyte page.

Because this default entry only translates a 4-Kbyte page, the initial code in this page needs to set up more valid TLB entries (and pages) so that the program can branch out of this

4-Kbyte page into other pages for booting the operating system. In particular, the interrupt vector area and the pages that contain the interrupt handlers should be set up so that exceptions can be handled early in the booting process.

Table 13-6. TLB1 Entry 0 Values after Reset

Field	Reset Value	Comments
V	1	Entry is valid
TS	0	Address space 0
TID[0–7]	0x00	TID value for shared (global) page
EPN[32–51]	0xFFFF	Address of last 4-Kbyte page in address space
RPN[32–51]	0xFFFF	Address of last 4-Kbyte page in address space
SIZE[0–3]	0001	4-Kbyte page size
SX/SR/SW	111	Full supervisor mode access allowed
UX/UR/UW	000	No user mode access allowed
WIMGE	01000	Caching-inhibited, non-coherent, big-endian
SHAREN	0	Read all lines as exclusive (3-state cache model)
X0–X1	00	Reserved system attributes
U0–U3	0000	User bits
IPROT	1	Page is protected from invalidation

13.7 Core Complex MMU Registers

Table 13-7 provides cross-references to other sections that have more detailed bit descriptions for the e500 registers related to the MMU.

Table 13-7. Registers Used for MMU Functions

Registers	Comprehensive Reference (Section/Page)	Motorola Book E Description (Section/Page)	Additional e500-Only Reference (Section/Page)
Process ID (PID0–PID2)	2.12.1/2-41	—	—
MMU control and status register (MMUCSR0)	2.12.2/2-41	—	—
MMU configuration register (MMUCFG)	2.12.3/2-42	11.3.7.1/11-49	—
TLB configuration registers (TLB0CFG–TLB1CFG)	2.12.4/2-43	11.3.7.2/11-49	—
MMU assist registers (MAS0–MAS4, MAS6)	2.12.5/2-45	11.3.5/11-38	13.7.1/13-24
Data exception address register (DEAR)	2.7.1.5/2-25	—	—

13.7.1 e500 MAS Registers

The core complex uses six special purpose registers (MAS0, MAS1, MAS2, MAS3, MAS4 and MAS6) to facilitate reading, writing, and searching the TLBs. The MAS registers can be read or written using the **mfsprr** and **mtsprr** instructions. The core complex does not implement the MAS5 register, because the **tlbsx** instruction on the e500 only searches based on a single PID value (the value of MAS6[SPID0]).

For the core complex, TLB0 is 2-way set associative, so bits 45–51 of the effective address are used to index into TLB0 when it is accessed. In this case, ESEL is only defined as one bit (bit 47) that identifies which of the two indexed entries is to be referenced by the TLB operation (ESEL selects the way). For TLB1, ESEL is a four-bit field that selects one of the 16 entries in the array.

Figure 13-8 describes the format of MAS0 on the e500 core complex.

	32	34	35	36		43	44		47	48		62	63
Field	—		TLBSEL		—				ESEL		—		NV
Reset	All zeros												
R/W	R/W												
SPR	SPR 624												

Figure 13-8. MAS Register 0 (MAS0) Format

Table 13-8 shows the core complex MAS0 bit definitions.

Table 13-8. MAS0 Field Descriptions—MMU Read/Write and Replacement Control

Bits	Name	Descriptions
32–34	—	Reserved, should be cleared.
35	TLBSEL	Selects TLB for access 0 TLB0 1 TLB1
36–43	—	Reserved, should be cleared.
44–47	ESEL	Entry select. Number of entry in selected array to be used for tlbwe . This field is also updated on TLB error exceptions (misses), and tlbsx hit and miss cases as shown in Table 13-14. For the e500, ESEL serves as the way select for the corresponding TLB as follows: When TLBSEL = 00 (TLB0 selected), only bit 47 is used (and bits 44–46 should be cleared). This bit selects between way 0 and way 1 of TLB0. EA bits 45–51 are used to index into the TLB to further select the entry for the operation. When TLBSEL = 01 (TLB1 selected), all four bits are used to select one of 16 entries in the array.
48–62	—	Reserved, should be cleared.
63	NV	Next victim. Next victim bit value to be written to TLB0[NV] on execution of tlbwe . This field is also updated on TLB error exceptions (misses), tlbsx hit and miss cases as shown in Table 13-14, and on execution of tlbre . This field is updated based on the calculated next victim bit for TLB0 (based on the round-robin replacement algorithm, described in Section 13.3.2.3, “Replacement Algorithms for L2 MMU”). Note that this field is not defined for operations that specify TLB1 (when TLBSEL = 01).

Figure 13-9 describes the format of MAS1 on the e500 core complex. Note that while the Motorola Book E allows for a TID field of 12 bits, the TID field on the core complex is implemented as only 8 bits.

	32	33	34		39	40		47	48	50	51	52		55	56		63
Field	V	IPROT	—		TID			—		TS	TSIZE		—				
Reset	All zeros																
R/W	R/W																
SPR	SPR 625																

Figure 13-9. MAS Register 1 (MAS1) Format

Table 13-9 shows the core complex MAS1 bit definitions.

Table 13-9. MAS1 Field Descriptions—Descriptor Context and Configuration Control

Bits	Name	Descriptions
32	V	TLB valid bit 0 This TLB entry is invalid. 1 This TLB entry is valid.
33	IPROT	Invalidate protect. Set to protect this TLB entry from invalidate operations due the execution of tlbiva[x] (TLB1 only). Note that not all TLB arrays are necessarily protected from invalidation with IPROT. Arrays that support invalidate protection are denoted as such in the TLB configuration registers. 0 Entry is not protected from invalidation 1 Entry is protected from invalidation. See Section 13.3.2.1, “IPROT Invalidation Protection in TLB1.”
34–39	—	Reserved, should be cleared.
40–47	TID	Translation identity. An 8-bit field that defines the process ID for this TLB entry. TID is compared with the current process IDs of the three virtual address to be translated. A TID value of 0 defines an entry as global and matches with all process IDs.
48–50	—	Reserved, should be cleared.
51	TS	Translation space. This bit is compared with the IS or DS fields of the MSR (depending on the type of access) to determine if this TLB entry may be used for translation.
52–55	TSIZE	Translation size. This 4-bit field defines the page size of the TLB entry. For TLB arrays that contain fixed-size TLB entries, this field is ignored. For variable page size TLB arrays, the page size is 4^{TSIZE} Kbytes. Note that the Motorola Book E standard supports all 16 page sizes defined in Book E.
56–63	—	Reserved, should be cleared.

Figure 13-10 describes the format of MAS2 on the e500 core complex. Note that the core complex defines the SHAREN bit not defined by the Motorola Book E standard.

	32	51	52	53	54	55	56	57	58	59	60	61	62	63	
Field	EPN				—	SHAREN		—	X0	X1	W	I	M	G	E
Reset	All zeros														
R/W	R/W														
SPR	SPR 626														

Figure 13-10. MAS Register 2 (MAS2) Format

Table 13-10 shows the core complex MAS2 bit definitions.

Table 13-10. MAS2 Field Descriptions—EPN and Page Attributes

Bits	Name	Description
32–51	EPN	Effective page number. Depending on page size, only the bits associated with a page boundary are valid. Bits that represent offsets within a page are ignored and should be cleared.
52–53	—	Reserved for implementation-specific use.
54	SHAREN	Enables cache fills to use shared cache state for this page. Note that this attribute is saved in the TLB entry when it is loaded. 0 Cache fills do not use the shared cache state for this page. 1 Cache fills use the shared cache state for this page.
55–56	—	Reserved, should be cleared.
57	X0	Implementation-dependent page attribute
58	X1	Implementation-dependent page attribute
59	W	Write-through 0 This page is considered write-back with respect to the caches in the system. 1 All stores performed to this page are written through the caches to main memory.
60	I	Caching-inhibited 0 Accesses to this page are considered cacheable. 1 The page is considered caching-inhibited. All loads and stores to the page bypass the caches and are performed directly to main memory.
61	M	Memory coherence required 0 Memory coherence is not required. 1 Memory coherence is required. This allows loads and stores to this page to be coherent with loads and stores from other processors (and devices) in the system, assuming all such devices are participating in the coherence protocol.
62	G	Guarded 0 Accesses to this page are not guarded and can be performed before it is known if they are required by the sequential execution model. 1 All loads and stores to this page that miss in the L1 cache are performed without speculation (that is, they are known to be required). Speculative loads can be performed if they hit in the L1 cache. In addition, accesses to caching-inhibited pages are performed using only the memory element that is explicitly specified.
63	E	Endianness. Determines endianness for the corresponding page. Little-endian operation is true little endian, which differs from the modified little-endian byte-ordering model optionally available in previous devices that implement the original PowerPC architecture. 0 The page is accessed in big-endian byte order. 1 The page is accessed in true little-endian byte order.

Table 13-12. MAS4 Field Descriptions—Hardware Replacement Assist Configuration

Bits	Name	Description
32–34	—	Reserved, should be cleared.
35	TLBSEL	TLBSEL default value. The default value to be loaded in MAS0[TLBSEL] on a TLB miss exception. See Section 11.3.6.1, “TLB Miss Exception Types,” for more information. 0 TLB0 1 TLB1
36–45	—	Reserved, should be cleared.
46–47	TIDSEL	TID default selection value. A 2-bit field that specifies which of the current PID registers should be used to load the MAS1[TID] field on a TLB miss exception. The e500 implementation defines this field as follows: 00 = PID0 01 = PID1 10 = PID2 11 = TIDZ (0x00) (all zeros)
48–50	—	Reserved, should be cleared.
51	SHAREND	Default SHAREN value
52–55	TSIZED	Default TSIZE value. Specifies the default value to be loaded into MAS1[TSIZE] on a TLB miss exception.
56	—	Reserved, should be cleared.
57	X0D	Default X0 value. Specifies the default value to be loaded into MAS2[X0] on a TLB miss exception.
58	X1D	Default X1 value. Specifies the default value to be loaded into MAS2[X1] on a TLB miss exception.
59	WD	Default W value. Specifies the default value to be loaded into MAS2[W] on a TLB miss exception.
60	ID	Default I value. Specifies the default value to be loaded into MAS2[I] on a TLB miss exception.
61	MD	Default M value. Specifies the default value to be loaded into MAS2[M] on a TLB miss exception.
62	GD	Default G value. Specifies the default value to be loaded into MAS2[G] on a TLB miss exception.
63	ED	Default E value. Specifies the default value to be loaded into MAS2[E] on a TLB miss exception.

Note that MAS5 is not implemented in the e500 core complex.

Table 13-13 shows the core complex MAS6 bit definitions. Note that while the Motorola Book E allows for a SPIDx field of 12 bits, SPID0 on the core complex is only an 8-bit field.

Table 13-13. MAS6—TLB Search Context Register 0

Bits	Name	Comments, or Function when Set
32–39	—	Reserved, should be cleared.
40–47	SPID0	Specifies the PID value (recent value of PID0) used when searching the TLB during execution of tlbsx .

Table 13-13. MAS6—TLB Search Context Register 0 (continued)

Bits	Name	Comments, or Function when Set
48–62	—	Reserved, should be cleared.
63	SAS	Address space (AS) value for searches. Specifies the value of AS used when searching the TLB (during execution of tlbsx).

13.7.2 MAS Register Updates

Table 13-14 summarizes the updates to each MAS register field for each update stimulus.

Table 13-14. MMU Assist Register Field Updates

MAS Register Bit/Field	Value Loaded for Each Case						
	Instr/Data TLB Error	tlbsx Hit	tlbsx Miss	ISI	DSI	tlbre	tlbwe
TLBSEL	TLBSELD	Which TLB hit	TLBSELD	—	—	—	—
ESEL	if TLBSELD = 0: TLB0[NV] else, undefined	Number of entry that hit	if TLBSELD = 0: TLB0[NV] else, undefined	—	—	—	—
NV	if TLBSELD = 0: ~TLB0[NV] else, undefined	if TLBSEL = 0: TLB0[NV] else, undefined	if TLBSELD = 0: ~TLB0[NV] else, undefined	—	—	if TLBSEL = 0: TLB0[NV] else, undefined	—
V	1	1	0	—	—	V(array)	—
IPROT	0	Matched IPROT if TLB1 hit; else 0	0	—	—	IPROT (array) if TLB1; else 0	—
TID[0–7]	Value of PID register selected by TIDSELD	TID (array)	SPID0	—	—	TID (array)	—
TS	MSR[IS/DS]	SAS	SAS	—	—	TS(array)	—
TSIZE[0–3]	TSIZED	TSIZE(array)	TSIZED	—	—	TSIZE(array)	—
EPN[32–51]	EPN of access	EPN (array)	—	—	—	EPN (array)	—
X0, X1 WIMGE	X0D, X1D WIMGED	X0, X1 (array) WIMGE (array)	X0D, X1D WIMGED	—	—	X0, X1 (array) WIMGE (array)	—
RPN[32–51]	Zeroes	RPN (array)	Zeroes	—	—	RPN (array)	—
PERMIS	Zeroes	PERMIS (array)	Zeroes	—	—	PERMIS (array)	—
TLBSELD	—	—	—	—	—	—	—
TIDSELD[0–1]	—	—	—	—	—	—	—
TSIZED[0–3]	—	—	—	—	—	—	—
WIMGED	—	—	—	—	—	—	—
SPID0	PID0	—	—	—	—	—	—

Table 13-14. MMU Assist Register Field Updates (continued)

MAS Register Bit/Field	Value Loaded for Each Case						
	Instr/Data TLB Error	tlbsx Hit	tlbsx Miss	ISI	DSI	tlbre	tlbwe
SAS	MSR[IS] for instruction access; MSR[DS] for data access	—	—	—	—	—	—
SHAREN	SHAREND	SHAREN (array)	SHAREND	—	—	SHAREN (array)	—

Appendix A

Simplified Mnemonics

This appendix describes simplified mnemonics that are provided for easier coding of assembly language programs. Simplified mnemonics are defined for the most frequently used forms of branch conditional, compare, trap, rotate and shift, and certain other instructions. (Note that the architecture specification refers to simplified mnemonics as extended mnemonics.)

A.1 Symbols

The symbols in Table A-1 are defined for use in instructions (basic or simplified mnemonics) that specify a condition register (CR) field or bit.

Table A-1. Condition Register Bit and Identification Symbol Descriptions

Symbol	Value	Bit Field Range ¹	Description
lt	0	—	Less than. Identifies a bit number within a CR field.
gt	1	—	Greater than. Identifies a bit number within a CR field.
eq	2	—	Equal. Identifies a bit number within a CR field.
so	3	—	Summary overflow. Identifies a bit number within a CR field.
cr0	0	32–35	CR0 field
cr1	1	36–39	CR1 field
cr2	2	40–43	CR2 field
cr3	3	44–47	CR3 field
cr4	4	48–51	CR4 field
cr5	5	52–55	CR5 field
cr6	6	56–59	CR6 field
cr7	7	60–63	CR7 field

¹ To identify a CR bit, an expression in which a CR field symbol is multiplied by 4 and then added to a bit-number-within-CR-field symbol can be used, (for example, cr0 * 4 + eq)

Note that the simplified mnemonics in Section A.5.2, “Basic Branch Mnemonics,” and Section A.6, “Simplified Mnemonics for Condition Register Logical Instructions,” require identification of a CR bit—if one of the CR field symbols is used, it must be multiplied by

4 and added to a bit-number-within-CR-field (value in the range of 0–3, explicit or symbolic).

The simplified mnemonics in Section A.5.3, “Branch Mnemonics Incorporating Conditions,” and Section A.3, “Simplified Mnemonics for Compare Instructions,” require identification of a CR field—if one of the CR field symbols is used, it must not be multiplied by 4. (For the simplified mnemonics in Section A.5.3, “Branch Mnemonics Incorporating Conditions,” the bit number within the CR field is part of the simplified mnemonic. The CR field is identified, and the assembler does the multiplication and addition required to produce a CR bit number for the BI field of the underlying basic mnemonic.)

A.2 Simplified Mnemonics for Subtract Instructions

This section discusses simplified mnemonics for the subtract instructions.

A.2.1 Subtract Immediate

Although there is no subtract immediate instruction, its effect can be achieved by using an add immediate instruction with the immediate operand negated. Simplified mnemonics are provided that include this negation, making the intent of the computation more clear.

Table A-2. Subtract Immediate Simplified Mnemonics

Simplified Mnemonic	Standard Mnemonic
subi rD,rA,value	addi rD,rA,-value
subis rD,rA,value	addis rD,rA,-value
subic rD,rA,value	addic rD,rA,-value
subic. rD,rA,value	addic. rD,rA,-value

A.2.2 Subtract

The subtract from instructions subtract the second operand (rA) from the third (rB). Simplified mnemonics are provided that use the more normal order in which the third operand is subtracted from the second. Both these mnemonics can be coded with an **o** suffix and/or dot (.) suffix to cause the OE and/or Rc bit to be set in the underlying instruction.

Table A-3. Subtract Immediate Simplified Mnemonics

Simplified Mnemonic	Standard Mnemonic
sub rD,rA,rB	subf rD,rB,rA
subc rD,rA,rB	subfc rD,rB,rA

Simplified Mnemonics for Rotate and Shift Instructions

- **Insert**—Select a left-justified or right-justified field of n bits in the source register; insert this field starting at bit position b of the target register; leave other bits of the target register unchanged. (No simplified mnemonic is provided for insertion of a left-justified field when operating on double words, because such an insertion requires more than one instruction.)
- **Rotate**—Rotate the contents of a register right or left n bits without masking.
- **Shift**—Shift the contents of a register right or left n bits, clearing vacated bits (logical shift).
- **Clear**—Clear the leftmost or rightmost n bits of a register.
- **Clear left and shift left**—Clear the leftmost b bits of a register, then shift the register left by n bits. This operation can be used to scale a (known non-negative) array index by the width of an element.

A.4.1 Operations on Words

The operations shown in Table A-5 are available in all implementations. These mnemonics can be coded with a dot (.) suffix to cause the Rc bit to be set in the underlying instruction.

Table A-5. Word Rotate and Shift Instructions

Operation	Simplified Mnemonic	Equivalent to:
Extract and left justify immediate	extlwi rA, rS, n, b ($n > 0$)	rlwinm $rA, rS, b, 0, n - 1$
Extract and right justify immediate	extrwi rA, rS, n, b ($n > 0$)	rlwinm $rA, rS, b + n, 32 - n, 31$
Insert from left immediate	inslwi rA, rS, n, b ($n > 0$)	rlwimi $rA, rS, 32 - b, b, (b + n) - 1$
Insert from right immediate	insrwi rA, rS, n, b ($n > 0$)	rlwimi $rA, rS, 32 - (b + n), b, (b + n) - 1$
Rotate left immediate	rotlwi rA, rS, n	rlwinm $rA, rS, n, 0, 31$
Rotate right immediate	rotrwi rA, rS, n	rlwinm $rA, rS, 32 - n, 0, 31$
Rotate left	rotlw rA, rS, rB	rlwnm $rA, rS, rB, 0, 31$
Shift left immediate	slwi rA, rS, n ($n < 32$)	rlwinm $rA, rS, n, 0, 31 - n$
Shift right immediate	srwi rA, rS, n ($n < 32$)	rlwinm $rA, rS, 32 - n, n, 31$
Clear left immediate	clrlwi rA, rS, n ($n < 32$)	rlwinm $rA, rS, 0, n, 31$
Clear right immediate	clrrwi rA, rS, n ($n < 32$)	rlwinm $rA, rS, 0, 0, 31 - n$
Clear left and shift left immediate	clrlslwi rA, rS, b, n ($n \leq b \leq 31$)	rlwinm $rA, rS, n, b - n, 31 - n$

Examples using word mnemonics follow:

1. Extract the sign bit (bit 0) of **rS** and place the result right-justified into **rA**.
extrwi rA,rS,1,0 equivalent to **rlwinm rA,rS,1,31,31**
2. Insert the bit extracted in (1) into the sign bit (bit 0) of **rB**.
insrwi rB,rA,1,0 equivalent to **rlwimi rB,rA,31,0,0**

3. Shift the contents of **rA** left 8 bits.
slwi rA,rA,8 equivalent to **rlwinm rA,rA,8,0,23**
4. Clear the high-order 16 bits of **rS** and place the result into **rA**.
clrlwi rA,rS,16 equivalent to **rlwinm rA,rS,0,16,31**

A.5 Simplified Mnemonics for Branch Instructions

Mnemonics are provided so that branch conditional instructions can be coded with the condition as part of the instruction mnemonic rather than as a numeric operand. Some of these are shown as examples with the branch instructions.

Mnemonics discussed in this section are variations of the branch conditional instructions.

A.5.1 BO and BI Fields

The 5-bit BO field in branch conditional instructions encodes the following operations.

- Decrement count register (CTR) and test if result is equal to zero
- Decrement CTR and test if result is not equal to zero
- Test condition true
- Test condition false
- Branch prediction (taken, fall through)

BO bit encodings are described in Table A-6,

Table A-6. BO Bit Encodings

BO Bit	Description
0	If set, ignore the CR bit comparison.
1	If set, the CR bit comparison is against “true”, if not set the CR bit comparison is against false
2	If set, the CTR is not decremented.
3	If BO[2] is set, this bit determines whether the CTR comparison is for equal to zero or not equal to zero.
4	If set, reverse the static prediction.

Thus, a BO encoding of 10100 (hex 0x14, decimal 20) means ignore the CR bit comparison, and don't decrement the CTR. In other words, branch unconditionally.

The 5-bit BI field in branch conditional instructions specifies which of the 32 bits in the CR represents the condition to test. For compatibility with AIM PowerPC applications, that CR bit selected is BI + 32, as shown in Table A-7.

Table A-7. BI Operand Settings for CR Fields

CRn Bits	CR Bits	BI	Description
CR0[0]	32	00000	Negative (LT)—Set when the result is negative.
CR0[1]	33	00001	Positive (GT)—Set when the result is positive (and not zero).
CR0[2]	34	00010	Zero (EQ)—Set when the result is zero.
CR0[3]	35	00011	Summary overflow (SO). Copy of XER[SO] at the instruction's completion.
CR1[0]	36	00100	—
CR1[1]	37	00101	—
CR1[2]	38	00110	—
CR1[3]	39	00111	—
CRn[0]	40 44 48 52 56 60	01000 01100 10000 10100 11000 11100	Less than (LT). For integer compare instructions: rA < SIMM or rB (signed comparison) or rA < UIMM or rB (unsigned comparison).
CRn[1]	41 45 49 53 57 61	01001 01101 10001 10101 11001 11101	Greater than (GT). For integer compare instructions: rA > SIMM or rB (signed comparison) or rA > UIMM or rB (unsigned comparison).
CRn[2]	42 46 50 54 58 62	01010 01110 10010 10110 11010 11110	Equal (EQ). For integer compare instructions: rA = SIMM, UIMM, or rB.
CRn[3]	43 47 51 55 59 63	01011 01111 10011 10111 11011 11111	Summary overflow (SO). For integer compare instructions, this is a copy of XER[SO] at the completion of the instruction.

To provide a simplified mnemonic for every possible combination of BO and BI fields would require $2^{10} = 1024$ mnemonics and most of these would be only marginally useful. The abbreviated set found in Section A.5.2, “Basic Branch Mnemonics,” is intended to cover the most useful cases. Unusual cases can be coded using a basic branch conditional mnemonic (**bc**, **bclr**, **bcctr**) with the condition to be tested specified as a numeric operand.

A.5.2 Basic Branch Mnemonics

The mnemonics in Table A-8 allow all the common BO operand encodings to be specified as part of the mnemonic, along with the absolute address (AA), and set link register (LK) bits.

Notice that there are no simplified mnemonics for relative and absolute unconditional branches. For these, the basic mnemonics **b**, **ba**, **bl**, and **bla** are used.

Table A-8 provides the abbreviated set of simplified mnemonics for the most commonly performed conditional branches.

Table A-8. Simplified Branch Mnemonics

Branch Semantics	LR Update Not Enabled				LR Update Enabled			
	bc Relative	bca Absolute	bclr to LR	bcctr to CTR	bcl Relative	bcla Absolute	bclrl to LR	bcctrl to CTR
Branch unconditionally	—	—	blr	bctr	—	—	blr	bctrl
Branch if condition true	bt	bta	btlr	btctr	btl	bta	btlr	btctrl
Branch if condition false	bf	bfa	bflr	bfctr	bfl	bfa	bflr	bfctrl
Decrement CTR, branch if CTR non-zero	bdnz	bdnza	bdnzlr	—	bdnzl	bdnzla	bdnzlr	—
Decrement CTR, branch if CTR non-zero AND condition true	bdnzt	bdnzta	bdnztlr	—	bdnztl	bdnzta	bdnztlr	—
Decrement CTR, branch if CTR non-zero AND condition false	bdnzf	bdnzfa	bdnzflr	—	bdnzfl	bdnzfa	bdnzflr	—
Decrement CTR, branch if CTR zero	bdz	bdza	bdzlr	—	bdzl	bdzla	bdzlr	—
Decrement CTR, branch if CTR zero AND condition true	bdzt	bdzta	bdztlr	—	bdztl	bdzta	bdztlr	—
Decrement CTR, branch if CTR zero AND condition false	bdzf	bdzfa	bdzflr	—	bdzfl	bdzfa	bdzflr	—

The simplified mnemonics shown in Table A-8 that test a condition require a corresponding CR bit as the first operand of the instruction. The symbols defined in Section A.1, “Symbols,” can be used in the operand in place of a numeric value.

The simplified mnemonics found in Table A-8 are used in the following examples:

- [illegible]

Table A-9 provides the simplified mnemonics for the **bc** and **bca** instructions without link register updating, and the syntax associated with these instructions. Note that the default CR bit (BI field) specified by the simplified mnemonics in the table is CR bit 0.

Table A-9. Simplified Mnemonics for bc and bca without LR Update

Branch Semantics	bc Relative	Simplified Mnemonic	bca Absolute	Simplified Mnemonic
Branch unconditionally	—	—	—	—
Branch if condition true	bc 12,0,target	bt 0,target	bca 12,0,target	bta 0,target
Branch if condition false	bc 4,0,target	bf 0,target	bca 4,0,target	bfa 0,target
Decrement CTR, branch if CTR nonzero	bc16,0,target	bdnz target	bca 16,0,target	bdnza target
Decrement CTR, branch if CTR nonzero AND condition true	bc 8,0,target	bdnzt 0,target	bca 8,0,target	bdnzta 0,target
Decrement CTR, branch if CTR nonzero AND condition false	bc 0,0,target	bdnzf 0,target	bca 0,0,target	bdnzfa 0,target
Decrement CTR, branch if CTR zero	bc18,0,target	bdz target	bca 18,0,target	bdza target
Decrement CTR, branch if CTR zero AND condition true	bc10,0,target	bdzt 0,target	bca 10,0,target	bdzta 0,target
Decrement CTR, branch if CTR zero AND condition false	bc 2,0,target	bdzf 0,target	bca 2,0,target	bdzfa 0,target

Table A-10 provides the simplified mnemonics for the **bclr** and **bcclr** instructions without link register updating, and the syntax associated with these instructions. Note that the default CR bit (BI field) specified by the simplified mnemonics in the table is CR bit 0.

Table A-10. Simplified Mnemonics for bclr and bcclr without LR Update

Branch Semantics	bclr to LR	Simplified Mnemonic	bcctr to CTR	Simplified Mnemonic
Branch unconditionally	bclr 20,0	blr	bcctr 20,0	bctr
Branch if condition true	bclr 12,0	btlr 0	bcctr 12,0	btctr 0
Branch if condition false	bclr 4,0	bflr 0	bcctr 4,0	bfctr 0
Decrement CTR, branch if CTR nonzero	bclr 16,0	bdnzlr	—	—
Decrement CTR, branch if CTR nonzero AND condition true	bclr 10,0	bdztlr 0	—	—
Decrement CTR, branch if CTR nonzero AND condition false	bclr 0,0	bdnzflr 0	—	—
Decrement CTR, branch if CTR zero	bclr 18,0	bdzlr	—	—
Decrement CTR, branch if CTR zero AND condition true	bclr 10,0	bdztlr 0	—	—
Decrement CTR, branch if CTR zero AND condition false	bclr 0,0	bdzflr 0	—	—

Table A-11 provides the simplified mnemonics for the **bcl** and **bcla** instructions with link register updating, and the syntax associated with these instructions. Note that the default CR bit (BI field) specified by the simplified mnemonics in the table is CR bit 0.

Table A-11. Simplified Mnemonics for bcl and bcla with LR Update

Branch Semantics	bcl Relative	Simplified Mnemonic	bcla Absolute	Simplified Mnemonic
Branch unconditionally	—	—	—	—
Branch if condition true	bcl 12,0,target	btl 0,target	bcla 12,0,target	btla 0,target
Branch if condition false	bcl 4,0,target	bfl 0,target	bcla 4,0,target	bfla 0,target
Decrement CTR, branch if CTR nonzero	bcl 16,0,target	bdnzl target	bcla 16,0,target	bdnzla target
Decrement CTR, branch if CTR nonzero AND condition true	bcl 8,0,target	bdnztl 0,target	bcla 8,0,target	bdnztla 0,target
Decrement CTR, branch if CTR nonzero AND condition false	bcl 0,0,target	bdnzfl 0,target	bcla 0,0,target	bdnzfla 0,target
Decrement CTR, branch if CTR zero	bcl 18,0,target	bdzl target	bcla 18,0,target	bdzla target
Decrement CTR, branch if CTR zero AND condition true	bcl 10,0,target	bdztl 0,target	bcla 10,0,target	bdztla 0,target
Decrement CTR, branch if CTR zero AND condition false	bcl 2,0,target	bdzfl 0,target	bcla 2,0,target	bdzfla 0,target

Table A-12 provides the simplified mnemonics for the **bclrl** and **bcctrl** instructions with link register updating, and the syntax associated with these instructions. Note that the default CR bit (BI field) specified by the simplified mnemonics in the table is CR bit 0.

Table A-12. Simplified Mnemonics for bclrl and bcctrl with LR Update

Branch Semantics	LR Update Enabled			
	bclrl to LR	Simplified Mnemonic	bcctrl to CTR	Simplified Mnemonic
Branch unconditionally	bclrl 20,0	blrl	bcctrl 20,0	bctrl
Branch if condition true	bclrl12,0	btlrl 0	bcctrl 12,0	btctrl 0
Branch if condition false	bclrl 4,0	bftrl 0	bcctrl 4,0	bfctrl 0
Decrement CTR, branch if CTR nonzero	bclrl 16,0	bdnzlrl	—	—
Decrement CTR, branch if CTR nonzero AND condition true	bclrl 8,0	bdnztrlrl 0	—	—
Decrement CTR, branch if CTR nonzero AND condition false	bclrl 0,0	bdnzflrl 0	—	—
Decrement CTR, branch if CTR zero	bclrl 18,0	bdzrlrl	—	—
Decrement CTR, branch if CTR zero AND condition true	bdztrlrl 0	bdztrlrl 0	—	—
Decrement CTR, branch if CTR zero AND condition false	bclrl 4,0	bftrl 0	—	—

A.5.3 Branch Mnemonics Incorporating Conditions

The mnemonics defined in Table A-14 are variations of the branch if condition true and branch if condition false BO encodings, with the most useful values of BI represented in the mnemonic rather than specified as a numeric operand.

A standard set of codes (shown in Table A-13) has been adopted for the most common combinations of branch conditions.

Table A-13. Standard Coding for Branch Conditions

Code	Description
lt	Less than
le	Less than or equal
eq	Equal
ge	Greater than or equal
gt	Greater than
nl	Not less than
ne	Not equal
ng	Not greater than
so	Summary overflow
ns	Not summary overflow

Table A-14 shows the simplified branch mnemonics incorporating conditions.

Table A-14. Simplified Mnemonics with Comparison Conditions

Branch Semantics	LR Update Not Enabled				LR Update Enabled			
	bc Relative	bca Absolute	bclr to LR	bcctr to CTR	bcl Relative	bcla Absolute	bclrl to LR	bcctrl to CTR
Branch if less than	blt	blta	bltlr	bltctr	bltl	bltla	bltlrl	bltctrl
Branch if less than or equal	ble	blea	blelr	blectr	blel	blela	blelrl	blectrl
Branch if equal	beq	beqa	beqlr	beqctr	beql	beqla	beqlrl	beqctrl
Branch if greater than or equal	bge	bgea	bgehr	bgectr	bgel	bgeha	bgehr	bgectrl
Branch if greater than	bgt	bgta	bgtlr	bgtctr	bgtl	bgtha	bgtlrl	bgtctrl
Branch if not less than	bnl	bnla	bnllr	bnlctr	bnll	bnlla	bnllrl	bnlctrl
Branch if not equal	bne	bnea	bnelr	bnectr	bnel	bnela	bnelrl	bnectrl
Branch if not greater than	bng	bnga	bnglr	bngctr	bngl	bngla	bnglrl	bngctrl
Branch if summary overflow	bsa	bsoa	bsolr	bsocctr	bsol	bsola	bsolrl	bsocctrl
Branch if not summary overflow	bns	bnsa	bnslr	bnsctr	bns	bnsa	bnsrl	bnsctrl

Table A-14. Simplified Mnemonics with Comparison Conditions (continued)

Branch Semantics	LR Update Not Enabled				LR Update Enabled			
	bc Relative	bca Absolute	bclr to LR	bcctr to CTR	bcl Relative	bcla Absolute	bclrl to LR	bcctrl to CTR
Branch if unordered	bun	buna	bunlr	bunctr	bunl	bunla	bunlrl	bunctrl
Branch if not unordered	bnv	bnva	bnvlr	bnvctr	bnvl	bnvla	bnvlrl	bnvctrl

Instructions using the mnemonics in Table A-14 specify the CR field in an optional first operand. If the CR field being tested is CR0, this operand need not be specified. One of the CR field symbols defined in Section A.1, “Symbols,” can be used for this operand.

The simplified mnemonics found in Table A-14 are used in the following examples:

1. Branch if CR0 reflects not-equal condition. bne target	equivalent to	bc 4,2,target
2. Same as (1) but condition is in CR3. bne cr3,target	equivalent to	bc 4,14,target
3. Branch to an absolute target if CR4 specifies greater than condition, setting the LR. This is a form of conditional call. bgla cr4,target	equivalent to	bcla 12,17,target
4. Same as (3), but target address is in the CTR. bgtctrl cr4	equivalent to	bcctrl 12,17

Table A-15 shows the simplified branch mnemonics for the **bc** and **bca** instructions without link register updating, and the syntax associated with these instructions. Note that the default CR bit (BI field) specified by the simplified mnemonics in the table is CR bit 0.

Table A-15. Simplified Mnemonics for bc and bca without Comparison Conditions or LR Updating

Branch Semantics	bc Relative	Simplified Mnemonic	bca Absolute	Simplified Mnemonic
Branch if less than	bc 12,0 ,target	blt target	bca 12,0 ,target	blta target
Branch if less than or equal	bc 4,1 ,target	ble target	bca 4,1 ,target	blea target
Branch if equal	bc 12,2 ,target	beq target	bca 12,2 ,target	beqa target
Branch if greater than or equal	bc 4,0 ,target	bge target	bca 4,0 ,target	bgea target
Branch if greater than	bc 12,1 ,target	bgt target	bca 12,1 ,target	bgta target
Branch if not less than	bc 4,0 ,target	bnl target	bca 4,0 ,target	bnla target
Branch if not equal	bc 4,2 ,target	bne target	bca 4,2 ,target	bn ea target
Branch if not greater than	bc 4,1 ,target	bng target	bca 4,1 ,target	bnga target
Branch if summary overflow	bc 12,3 ,target	bs o target	bca 12,3 ,target	bs oa target
Branch if not summary overflow	bc 4,3 ,target	bns target	bca 4,3 ,target	bns a target

Table A-15. Simplified Mnemonics for bc and bca without Comparison Conditions or LR Updating

Branch Semantics	bc Relative	Simplified Mnemonic	bca Absolute	Simplified Mnemonic
Branch if unordered	bc 12,3,target	bun target	bca 12,3,target	buna target
Branch if not unordered	bc 4,3,target	bnu target	bca 4,3,target	bnua target

Table A-16 shows the simplified branch mnemonics for the **bclr** and **bcctr** instructions without link register updating, and the syntax associated with these instructions. Note that the default CR bit (BI field) specified by the simplified mnemonics in the table is CR bit 0.

Table A-16. Simplified Mnemonics for bclr and bcctr without Comparison Conditions and LR Updating

Branch Semantics	bclr to LR	Simplified Mnemonic	bcctr to CTR	Simplified Mnemonic
Branch if less than	bclr 12,0	bltlr	bcctr 12,0	bltctr
Branch if less than or equal	bclr 4,1	blelr	bcctr 4,1	blectr
Branch if equal	bclr 12,2	beqlr	bcctr 12,2	beqctr
Branch if greater than or equal	bclr 4,0	bgehr	bcctr 4,0	bgectr
Branch if greater than	bclr 12,1	bgthl	bcctr 12,1	bgctr
Branch if not less than	bclr 4,0	bnllr	bcctr 4,0	bnlctr
Branch if not equal	bclr 4,2	bnelr	bcctr 4,2	bnctr
Branch if not greater than	bclr 4,1	bnghl	bcctr 4,1	bnghctr
Branch if summary overflow	bclr 12,3	bsolr	bcctr 12,3	bsocctr
Branch if not summary overflow	bclr 4,3	bsnlr	bcctr 4,3	bsncctr
Branch if unordered	bclr 12,3	bunlr	bcctr 12,3	bunctr
Branch if not unordered	bclr 4,3	bnulr	bcctr 4,3	bnucctr

Table A-17 shows the simplified branch mnemonics for the **bcl** and **bcla** instructions with link register updating, and the syntax associated with these instructions. Note that the default CR bit (BI field) specified by the simplified mnemonics in the table is CR bit 0.

Table A-17. Simplified Mnemonics for bcl and bcla with Comparison Conditions and LR Updating

Branch Semantics	bcl Relative	Simplified Mnemonic	bcla Absolute	Simplified Mnemonic
Branch if less than	bcl 12,0,target	bltl target	bcla 12,0,target	bltla target
Branch if less than or equal	bcl 4,1,target	blel target	bcla 4,1,target	blela target
Branch if equal	bcl target	beql target	bcla 12,2,target	beqla target
Branch if greater than or equal	bcl 4,0,target	bgehr target	bcla 4,0,target	bgela target

Table A-17. Simplified Mnemonics for bcl and bcla with Comparison Conditions and LR Updating

Branch Semantics	bcl Relative	Simplified Mnemonic	bcla Absolute	Simplified Mnemonic
Branch if greater than	bcl 12,1,target	bgtl target	bcla 12,1,target	bgsla target
Branch if not less than	bcl 4,0,target	bnll target	bcla 4,0,target	bnlla target
Branch if not equal	bcl 4,2,target	bnel target	bcla 4,2,target	bnela target
Branch if not greater than	bcl 4,1,target	bngl target	bcla 4,1,target	bngla target
Branch if summary overflow	bcl 12,3,target	bsol target	bcla 12,3,target	bsola target
Branch if not summary overflow	bcl 4,3,target	bsnl target	bcla 4,3,target	bsnla target
Branch if unordered	bcl 12,3,target	bunl target	bcla 12,3,target	bunla target
Branch if not unordered	bcl 4,3,target	bnul target	bcla 4,3,target	bnula target

Table A-18 shows the simplified branch mnemonics for the **bclrl** and **bcctrl** instructions with link register updating, and the syntax associated with these instructions. Note that the default CR bit (BI field) specified by the simplified mnemonics in the table is CR bit 0.

Table A-18. Simplified Mnemonics for bclrl and bcctrl with Comparison Conditions and LR Update

Branch Semantics	bclrl to LR	Simplified Mnemonic	bcctrl to CTR	Simplified Mnemonic
Branch if less than	bclrl 12,0	bltlrl 0	bcctrl 12,0	bltctrl 0
Branch if less than or equal	bclrl 4,1	blelrl 0	bcctrl 4,1	blectrl 0
Branch if equal	bclrl 12,2	beqlrl 0	bcctrl 12,2	beqctrl 0
Branch if greater than or equal	bclrl 4,0	bgeqlrl 0	bcctrl 4,0	bgectrl 0
Branch if greater than	bclrl 12,1	bgtrlrl 0	bcctrl 12,1	bgctrl 0
Branch if not less than	bclrl 4,0	bnllrl 0	bcctrl 4,0	bnlctrl 0
Branch if not equal	bclrl 4,2	bnelrl 0	bcctrl 4,2	bnctrl 0
Branch if not greater than	bclrl 4,1	bnglrl 0	bcctrl 4,1	bngctrl 0
Branch if summary overflow	bclrl 12,3	bsolrl 0	bcctrl 12,3	bsocctrl 0
Branch if not summary overflow	bclrl 4,3	bsnlrl 0	bcctrl 4,3	bsncctrl 0
Branch if unordered	bclrl 12,3	bunlrl 0	bcctrl 12,3	buncctrl 0
Branch if not unordered	bclrl 4,3	bnulrl 0	bcctrl 4,3	bnucctrl 0

A.5.4 Branch Prediction

In branch conditional instructions that are not always taken, the low-order bit (y bit) of the BO field provides a hint about whether the branch is likely to be taken. See Section 3.4.1.4, “Conditional Branch Control,” for more information on the y bit.

Assemblers should clear this bit unless otherwise directed. This default action indicates the following:

- A branch conditional with a negative displacement field is predicted to be taken.
- A branch conditional with a non-negative displacement field is predicted not to be taken (fall through).
- A branch conditional to an address in the LR or CTR is predicted not to be taken (fall through).

If the likely outcome (branch or fall through) of a given branch conditional instruction is known, a suffix can be added to the mnemonic that tells the assembler how to set the y bit. That is, ‘+’ indicates that the branch is to be taken and ‘-’ indicates that the branch is not to be taken. Such a suffix can be added to any branch conditional mnemonic, either basic or simplified.

For relative and absolute branches (**bc[l][a]**), the setting of the y bit depends on whether the displacement field is negative or non-negative. For negative displacement fields, coding the suffix ‘+’ causes the bit to be cleared, and coding the suffix ‘-’ causes the bit to be set. For non-negative displacement fields, coding the suffix ‘+’ causes the bit to be set, and coding the suffix ‘-’ causes the bit to be cleared.

For branches to an address in the LR or CTR (**bcclr[l]** or **bcctr[l]**), coding the suffix ‘+’ causes the y bit to be set, and coding the suffix ‘-’ causes the bit to be cleared.

Examples of branch prediction follow:

1. Branch if CR0 reflects less than condition, specifying that the branch should be predicted to be taken.
blt+ target
2. Same as (1), but target address is in the LR and the branch should be predicted not to be taken.
bldlr-

A.6 Simplified Mnemonics for Condition Register Logical Instructions

The CR logical instructions, shown in Table A-19, can be used to set, clear, copy, or invert a given CR bit. Simplified mnemonics allow these operations to be coded easily. Note that the symbols defined in Section A.1, “Symbols,” can be used to identify the CR bit.

Table A-19. Condition Register Logical Mnemonics

Operation	Simplified Mnemonic	Equivalent to
Condition register set	crset bx	creqv bx,bx,bx
Condition register clear	crclr bx	crxor bx,bx,bx

Table A-19. Condition Register Logical Mnemonics (continued)

Operation	Simplified Mnemonic	Equivalent to
Condition register move	crmove bx,by	cror bx,by,by
Condition register not	crnot bx,by	crnor bx,by,by

Examples using the CR logical mnemonics follow:

1. Set CR[57].
crset 25 equivalent to **creqv 25,25,25**
2. Clear CR0[SO].
crclr so equivalent to **crxor 3,3,3**
3. Same as (2), but clear CR3[SO].
crclr 4 * cr3 + so equivalent to **crxor 15,15,15**
4. Invert the EQ bit.
crnot eq,eq equivalent to **crnor 2,2,2**
5. Same as (4), but CR4[EQ] is inverted and the result is placed into CR5[EQ].
crnot 4 * cr5 + eq, 4 * cr4 + eq equivalent to **crnor 22,18,18**

A.7 Simplified Mnemonics for Trap Instructions

A standard set of codes, shown in Table A-20, has been adopted for the most common combinations of trap conditions.

Table A-20. Standard Codes for Trap Instructions

Code	Description	TO Encoding	<	>	=	<U ¹	>U ²
lt	Less than	16	1	0	0	0	0
le	Less than or equal	20	1	0	1	0	0
eq	Equal	4	0	0	1	0	0
ge	Greater than or equal	12	0	1	1	0	0
gt	Greater than	8	0	1	0	0	0
nl	Not less than	12	0	1	1	0	0
ne	Not equal	24	1	1	0	0	0
ng	Not greater than	20	1	0	1	0	0
llt	Logically less than	2	0	0	0	1	0
lle	Logically less than or equal	6	0	0	1	1	0
lge	Logically greater than or equal	5	0	0	1	0	1
lgt	Logically greater than	1	0	0	0	0	1
lnl	Logically not less than	5	0	0	1	0	1

Table A-20. Standard Codes for Trap Instructions (continued)

Code	Description	TO Encoding	<	>	=	<U ¹	>U ²
Ing	Logically not greater than	6	0	0	1	1	0
—	Unconditional	31	1	1	1	1	1

¹ The symbol '<U' indicates an unsigned less than evaluation is performed.

² The symbol '>U' indicates an unsigned greater than evaluation is performed.

The mnemonics defined in Table A-21 are variations of trap instructions, with the most useful TO values represented in the mnemonic rather than specified as a numeric operand.

Table A-21. Trap Mnemonics

Trap Semantics	32-Bit Comparison	
	twi Immediate	tw Register
Trap unconditionally	—	trap
Trap if less than	twlti	twlt
Trap if less than or equal	twlei	twle
Trap if equal	tweqi	tweq
Trap if greater than or equal	twgei	twge
Trap if greater than	twgti	twgt
Trap if not less than	twnli	twnl
Trap if not equal	twnei	twne
Trap if not greater than	twngi	twng
Trap if logically less than	twllti	twllt
Trap if logically less than or equal	twlle	twlle
Trap if logically greater than or equal	twlgei	twlge
Trap if logically greater than	twlgti	twlgt
Trap if logically not less than	twlnli	twlnl
Trap if logically not greater than	twlngi	twlng

Examples of the uses of trap mnemonics, shown in Table A-21 follow:

1. Trap if register rA is not zero. twnei rA,0	equivalent to	twi 24,rA,0
2. Trap if register rA is not equal to rB . twne rA, rB	equivalent to	tw 24,rA,rB
3. Trap if rA is logically greater than 0x7FF. twlgti rA, 0x7FF	equivalent to	twi 1,rA, 0x7FF

4. Trap unconditionally.

trap equivalent to **tw 31,0,0**

Trap instructions evaluate a trap condition as follows:

- The contents of register **rA** are compared with either the sign-extended SIMM field or the contents of register **rB**, depending on the trap instruction.

The comparison results in five conditions which are ANDed with operand **TO**. If the result is not 0, the trap exception handler is invoked. (Note that exceptions are referred to as interrupts in the architecture specification.) See Table A-22 for these conditions.

Table A-22. TO Operand Bit Encoding

TO Bit	ANDed with Condition
0	Less than, using signed comparison
1	Greater than, using signed comparison
2	Equal
3	Less than, using unsigned comparison
4	Greater than, using unsigned comparison

A.8 Simplified Mnemonics for SPRs

The **mtspr** and **mfspir** instructions specify a special-purpose register (SPR) as a numeric operand. Simplified mnemonics are provided that represent the SPR in the mnemonic rather than requiring it to be coded as a numeric operand. Table A-23 lists simplified mnemonics for accessing Book E SPRs.

Table A-23. Simplified Mnemonics for Accessing Book E SPRs

SPR	Move from SPR		Move to SPR	
	Simplified Mnemonic	Equivalent to	Simplified Mnemonic	Equivalent to
Critical save/restore register 0	mfcsrr0 rD	mfspir 58, rD	mtcsrr0 rS	mtspir 58, rS
Critical save/restore register 1	mfcsrr1 rD	mfspir 59, rD	mtcsrr1 rS	mtspir 59, rS
Count register	mfctr rD	mfspir 9, rD	mtctr rS	mtspir 9, rS
Data address compare 1	mfdac1 rD	mfspir 316, rD	mtdac1 rS	mtspir 316, rS
Data address compare 2	mfdac2 rD	mfspir 317, rD	mtdac2 rS	mtspir 317, rS
Debug control register 0	mfdbcr0 rD	mfspir 308, rD	mtdbcr0 rS	mtspir 308, rS
Debug control register 1	mfdbcr1 rD	mfspir 309, rD	mtdbcr1 rS	mtspir 309, rS
Debug control register 2	mfdbcr2 rD	mfspir 310, rD	mtdbcr2 rS	mtspir 310, rS
Debug status register	mfdbsr rD	mfspir 304, rD	mtdbsr rS	mtspir 304, rS
Data exception address register	mfdear rD	mfspir 61, rD	mtdear rS	mtspir 61, rS
Decrementer	mfdec rD	mfspir 22, rD	mtdec rS	mtspir 22, rS

Table A-23. Simplified Mnemonics for Accessing Book E SPRs

SPR	Move from SPR		Move to SPR	
	Simplified Mnemonic	Equivalent to	Simplified Mnemonic	Equivalent to
Decrementer auto-reload	—	—	mtdecar rS	mtspr 54, rS
Data value compare 1	mfdvc1 rD	mfspir 318, rD	mtdvc1 rS	mtspr 318, rS
Data value compare 2	mfdvc2 rD	mfspir 319, rD	mtdvc2 rS	mtspr 319, rS
Exception syndrome register	mfesr rD	mfspir 62, rD	mtesr rS	mtspr 62, rS
Interrupt vector prefix register	mfivpr rD	mfspir 63, rD	mtivpr rS	mtspr 63, rS
Instruction address compare 1	mfiacl rD	mfspir 312, rD	mtiac1 rS	mtspr 312, rS
Instruction address compare 2	mfiacl rD	mfspir 313, rD	mtiac2 rS	mtspr 313, rS
Instruction address compare 3	mfiacl rD	mfspir 314, rD	mtiac3 rS	mtspr 314, rS
Instruction address compare 4	mfiacl rD	mfspir 315, rD	mtiac4 rS	mtspr 315, rS
Interrupt vector offset register 0	mfivor0 rD	mfspir 400, rD	mtivor0 rS	mtspr 400, rS
Interrupt vector offset register 1	mfivor1 rD	mfspir 401, rD	mtivor1 rS	mtspr 401, rS
Interrupt vector offset register 2	mfivor2 rD	mfspir 402, rD	mtivor2 rS	mtspr 402, rS
Interrupt vector offset register 3	mfivor3 rD	mfspir 403, rD	mtivor3 rS	mtspr 403, rS
Interrupt vector offset register 4	mfivor4 rD	mfspir 404, rD	mtivor4 rS	mtspr 404, rS
Interrupt vector offset register 5	mfivor5 rD	mfspir 405, rD	mtivor5 rS	mtspr 405, rS
Interrupt vector offset register 6	mfivor6 rD	mfspir 406, rD	mtivor6 rS	mtspr 406, rS
Interrupt vector offset register 7	mfivor7 rD	mfspir 407, rD	mtivor7 rS	mtspr 407, rS
Interrupt vector offset register 8	mfivor8 rD	mfspir 408, rD	mtivor8 rS	mtspr 408, rS
Interrupt vector offset register 9	mfivor9 rD	mfspir 409, rD	mtivor9 rS	mtspr 409, rS
Interrupt vector offset register 10	mfivor10 rD	mfspir rD, 410	mtivor10 rS	mtspr 410, rS
Interrupt vector offset register 11	mfivor11 rD	mfspir rD, 411	mtivor11 rS	mtspr 411, rS
Interrupt vector offset register 12	mfivor12 rD	mfspir rD, 412	mtivor12 rS	mtspr 412, rS
Interrupt vector offset register 13	mfivor13 rD	mfspir rD, 413	mtivor13 rS	mtspr 413, rS
Interrupt vector offset register 14	mfivor14 rD	mfspir rD, 414	mtivor14 rS	mtspr 414, rS
Interrupt vector offset register 15	mfivor15 rD	mfspir rD, 415	mtivor15 rS	mtspr 415, rS
Link register	mfllr rD	mfspir rD, 8	mtllr rS	mtspr 8, rS
Process ID register ¹	mfpid rD	mfspir rD, 48	mtpid rS	mtspr 48, rS
Processor ID register	mfpir rD	mfspir rD, 286	—	—
Processor version register	mfpir rD	mfspir rD, 287	—	—
SPR general 0	mfspgr0 rD	mfspir rD, 272	mtspgr0 rS	mtspr 272, rS
SPR general 1	mfspgr1 rD	mfspir rD, 273	mtspgr1 rS	mtspr 273, rS
SPR general 2	mfspgr2 rD	mfspir rD, 274	mtspgr2 rS	mtspr 274, rS
SPR general 3 (user-level)	mfspgr3 rD	mfspir rD, 259	—	—
SPR general 3 (supervisor-level)	mfspgr3 rD	mfspir rD, 275	mtspgr3 rS	mtspr 275, rS

Table A-23. Simplified Mnemonics for Accessing Book E SPRs

SPR	Move from SPR		Move to SPR	
	Simplified Mnemonic	Equivalent to	Simplified Mnemonic	Equivalent to
SPR general 4 (user-level)	mfsp ^{rg} 4 rD	mf ^{spr} rD, 260	—	—
SPR general 4 (supervisor-level)	mfsp ^{rg} 4 rD	mf ^{spr} rD, 276	mts ^{prg} 4 rS	mts ^{pr} 276, rS
SPR general 5 (user-level)	mfsp ^{rg} 5 rD	mf ^{spr} rD, 261	—	—
SPR general 5 (supervisor-level)	mfsp ^{rg} 5 rD	mf ^{spr} rD, 277	mts ^{prg} 5 rS	mts ^{pr} 277, rS
SPR general 6 (user-level)	mfsp ^{rg} 6 rD	mf ^{spr} rD, 262	—	—
SPR general 6 (supervisor-level)	mfsp ^{rg} 6 rD	mf ^{spr} rD, 278	mts ^{prg} 6 rS	mts ^{pr} 278, rS
SPR general 7 (user-level)	mfsp ^{rg} 7 rD	mf ^{spr} rD, 263	—	—
SPR general 7 (supervisor-level)	mfsp ^{rg} 7 rD	mf ^{spr} rD, 279	mts ^{prg} 7 rS	mts ^{pr} 279, rS
User SPR general 0	mfus ^{prg} 0 rD	mf ^{spr} rD, 256	mtus ^{prg} 0 rS	mts ^{pr} 256, rS
Save/restore register 0	mfs ^{rr} 0 rD	mf ^{spr} rD, 26	mts ^{rr} 0 rS	mts ^{pr} 26, rS
Save/restore register 1	mfs ^{rr} 1 rD	mf ^{spr} rD, 27	mts ^{rr} 1 rS	mts ^{pr} 27, rS
Time base lower (user/supervisor read)	mftbl rD	mf ^{spr} rD, 268	—	—
Time base lower (supervisor write)	—	—	mttbl rS	mts ^{pr} 284, rS
Time base upper (user/supervisor read)	mftbu rD	mf ^{spr} rD, 269	—	—
Time base upper (supervisor write)	—	—	mttbu rS	mts ^{pr} 285, rS
Timer control register	mftcr rD	mf ^{spr} rD, 340	mttcr rS	mts ^{pr} 340, rS
Timer status register	mftsr rD	mf ^{spr} rD, 336	mttsr rS	mts ^{pr} 336, rS
Integer exception register	mfxer rD	mf ^{spr} rD, 1	mtxer rS	mts ^{pr} 1, rS

¹ Book E defines only one PID, which is designated as PID0 for Motorola multiple-PID implementations.

Table A-24 lists simplified mnemonics for accessing e500-specific SPRs.

Table A-24. Simplified Mnemonics for Accessing e500-Specific SPR

SPR	Move from SPR		Move to SPR	
	Simplified Mnemonic	Equivalent to	Simplified Mnemonic	Equivalent to
L1 cache configuration register 0	mfl1c ^{fg} 0 rD	mf ^{spr} rD, 515	—	—
L1 cache configuration register 1	mfl1c ^{fg} 1 rD	mf ^{spr} rD, 516	—	—
TLB configuration register 0	mftlb0c ^{fg} rD	mf ^{spr} rD, 688	—	—
TLB configuration register 1	mftlb1c ^{fg} rD	mf ^{spr} rD, 689	—	—
MMU configuration register	mfm ^{muc} fg rD	mf ^{spr} rD, 1015	—	—

Table A-24. Simplified Mnemonics for Accessing e500-Specific SPR

SPR	Move from SPR		Move to SPR	
	Simplified Mnemonic	Equivalent to	Simplified Mnemonic	Equivalent to
Signal processing and embedded floating-point status and control register	mfspefscr rD	mfspr rD, 512	mtspefscr rS	mtspr 512, rS
Branch buffer entry address register	mfbbear rD	mfspir rD, 513	mtbbear rS	mtspr 513, rS
Branch buffer target address register	mfbbtar rD	mfspir rD, 514	mtbbtar rS	mtspr 514, rS
Nexus processor ID register	mfnpidr rD	mfspir rD, 517	mtnpidr rS	mtspr 517, rS
Instruction address resume register	mfiarr rD	mfspir rD, 518	mtiarr rS	mtspr 518, rS
Interrupt vector offset register 32 (SPU interrupt)	mfivor32 rD	mfspir rD, 528	mtivor32 rS	mtspr 528, rS
Interrupt vector offset register 33 (Vector floating-point data exception)	mfivor33 rD	mfspir rD, 529	mtivor33 rS	mtspr 529, rS
Interrupt vector offset register 34 (Vector floating-point round exception)	mfivor34 rD	mfspir rD, 530	mtivor34 rS	mtspr 530, rS
Interrupt vector offset register 35 (performance monitor)	mfivor35 rD	mfspir rD, 531	mtivor35 rS	mtspr 531, rS
Machine-check save/restore register 0	mfmcrr0 rD	mfspir rD, 570	mtmcrr0 rS	mtspr 570, rS
Machine-check save/restore register 1	mfmcrr1 rD	mfspir rD, 571	mtmcrr1 rS	mtspr 571, rS
Machine check syndrome register	mfmcrr rD	mfspir rD, 572	mtmcrr rS	mtspr 572, rS
MMU assist register 0	mfmas0 rD	mfspir rD, 624	mtmas0 rS	mtspr 624, rS
MMU assist register 1	mfmas1 rD	mfspir rD, 625	mtmas1 rS	mtspr 625, rS
MMU assist register 2	mfmas2 rD	mfspir rD, 626	mtmas2 rS	mtspr 626, rS
MMU assist register 3	mfmas3 rD	mfspir rD, 627	mtmas3 rS	mtspr 627, rS
MMU assist register 4	mfmas4 rD	mfspir rD, 628	mtmas4 rS	mtspr 628, rS
MMU assist register 5 Defined by Motorola Book E implementation standard but not implemented in the e500	mfmas5 rD	mfspir rD, 629	mtmas5 rS	mtspr 629, rS
MMU assist register 6	mfmas6 rD	mfspir rD, 630	mtmas6 rS	mtspr 630, rS
Process ID register 0. Book E defines only this PID register and refers to as PID rather than PID0.	mfpid0 rD	mfspir rD, 48	mtpid0 rS	mtspr 48, rS
Process ID register 1	mfpid1 rD	mfspir rD, 633	mtpid1 rS	mtspr 633, rS
Process ID register 2	mfpid2 rD	mfspir rD, 634	mtpid2 rS	mtspr 634, rS

Table A-24. Simplified Mnemonics for Accessing e500-Specific SPR

SPR	Move from SPR		Move to SPR	
	Simplified Mnemonic	Equivalent to	Simplified Mnemonic	Equivalent to
Hardware implementation dependent reg 0	mfhid0 rD	mfspr rD, 1008	mthid0 rS	mtspr 1008, rS
Hardware implementation dependent reg 1	mfhid1 rD	mfspr rD, 1009	mthid1 rS	mtspr 1009, rS
L1 cache control and status register 0	mf1csr0 rD	mfspr rD, 1010	mtl1csr0 rS	mtspr 1010, rS
L1 cache control and status register 1	mf1csr1 rD	mfspr rD, 1011	mtl1csr1 rS	mtspr 1011, rS
MMU control and status register 0	mfmmucsr0 rD	mfspr rD, 1012	mtmmucsr0 rS	mtspr 1012, rS
Branch unit control and status register 0	mfbucsr rD	mfspr rD, 1013	mtbucsr rS	mtspr 1013, rS

Following are examples using the SPR simplified mnemonics found in Table A-22:

1. Copy the contents of rS to the XER. mtxer rS	equivalent to	mtspr 1,rS
2. Copy the contents of the LR to rS . mflr rS	equivalent to	mfspr rS,8
3. Copy the contents of rS to the CTR. mtctr rS	equivalent to	mtspr 9,rS

A.9 Recommended Simplified Mnemonics

This section describes some of the most commonly-used operations (such as no-op, load immediate, load address, move register, and complement register).

A.9.1 No-Op (nop)

Many instructions can be coded in a way that, effectively, no operation is performed. An additional mnemonic is provided for the preferred form of no-op. If an implementation performs any type of run-time optimization related to no-ops, the preferred form is the no-op that triggers the following:

nop equivalent to **ori 0,0,0**

Note that on the e500, the **nop** instruction works very efficiently in that it is not dispatched to an execution unit.

A.9.2 Load Immediate (li)

The **addi** and **addis** instructions can be used to load an immediate value into a register. Additional mnemonics are provided to convey the idea that no addition is being performed but that data is being moved from the immediate operand of the instruction to a register.

1. Load a 16-bit signed immediate value into **rD**.
li rD,value equivalent to **addi rD,0,value**
2. Load a 16-bit signed immediate value, shifted left by 16 bits, into **rD**.
lis rD,value equivalent to **addis rD,0,value**

A.9.3 Load Address (la)

This mnemonic permits computing the value of a base-displacement operand, using the **addi** instruction which normally requires a separate register and immediate operands.

la rD,d(rA) equivalent to **addi rD,rA,d**

The **la** mnemonic is useful for obtaining the address of a variable specified by name, allowing the assembler to supply the base register number and compute the displacement. If the variable **v** is located at offset **dV** bytes from the address in register **rV**, and the assembler has been told to use register **rV** as a base for references to the data structure containing **v**, the following line causes the address of **v** to be loaded into register **rD**:

la rD,v equivalent to **addi rD,rV,dv**

A.9.4 Move Register (mr)

Several instructions can be coded to copy the contents of one register to another. A simplified mnemonic is provided that signifies that no computation is being performed, but merely that data is being moved from one register to another.

The following instruction copies the contents of **rS** into **rA**. This mnemonic can be coded with a dot (.) suffix to cause the Rc bit to be set in the underlying instruction.

mr rA,rS equivalent to **or rA,rS,rS**

A.9.5 Complement Register (not)

Several instructions can be coded in a way that they complement the contents of one register and place the result into another register. A simplified mnemonic is provided that allows this operation to be coded easily.

The following instruction complements the contents of **rS** and places the result into **rA**. This mnemonic can be coded with a dot (.) suffix to cause the Rc bit to be set in the underlying instruction.

not rA,rS equivalent to **nor rA,rS,rS**

A.9.6 Move to Condition Register (mtcr)

This mnemonic permits copying the contents of a GPR to the CR, using the same syntax as the **mfcrr** instruction.

mtcr rS equivalent to **mtcrr 0xFF,rS**

A.10 e500-Specific Simplified Mnemonics

A.10.1 Integer Select (isel)

The following mnemonics simplify the use of the most common variants of the **isel** instruction:

Integer Select Less Than

isellt rD,rA,rB equivalent to **isel rD,rA,rB,0**

Integer Select Greater Than

iselgt rD,rA,rB equivalent to **isel rD,rA,rB,1**

Integer Select Equal

iseleq rD,rA,rB equivalent to **isel rD,rA,rB,2**

A.10.2 SPU Mnemonics

The following mnemonic handles moving of the full 64-bit SPU register.

evmr rD,rA equivalent to **evor rD,rA,rA**

The following mnemonic performs a complement register

evnot rD,rA equivalent to **evnor rD,rA,rA**

A.11 Complete List of Simplified Mnemonics

Table A-25 lists simplified mnemonics that are supported by the e500 processor. Note that compiler designers may implement additional simplified mnemonics that are not listed here.

Table A-25. Simplified Mnemonics

Simplified Mnemonic	Mnemonic	Instruction
bctr	bcctr 20,0	Branch unconditionally (bcctr to CTR)
bctrl	bcctrl 20,0	Branch unconditionally (bcctrl to CTR)
bdnz target	bc 16,0,target	Decrement CTR, branch if CTR nonzero (bc relative)

Table A-25. Simplified Mnemonics (continued)

Simplified Mnemonic	Mnemonic	Instruction
bdnza target	bca 16,0,target	Decrement CTR, branch if CTR nonzero (bca Absolute)
bdnzf 0,target	bc 0,0,target	Decrement CTR, branch if CTR nonzero AND condition false (bc relative)
bdnzfa 0,target	bca 0,0,target	Decrement CTR, branch if CTR nonzero AND condition false (bca absolute)
bdnzfl 0,target	bcl 0,0,target	Decrement CTR, branch if CTR nonzero AND condition false (bcl relative)
bdnzfla 0,target	bcla 0,0,target	Decrement CTR, branch if CTR nonzero AND condition false (bcla absolute)
bdnzflr 0	bclr 0,0	Decrement CTR, branch if CTR nonzero AND condition false (bclr to LR)
bdnzflrl 0	bclrl 0,0	Decrement CTR, branch if CTR nonzero AND condition false (bclrl to LR)
bdnzl target	bcl 16,0,target	Decrement CTR, branch if CTR nonzero (bcl relative)
bdnzla target	bcla 16,0,target	Decrement CTR, branch if CTR nonzero (bcla absolute)
bdnzlr	bclr 16,0	Decrement CTR, branch if CTR nonzero (bclr to LR)
bdnzlrl	bclrl 16,0	Decrement CTR, branch if CTR nonzero (bclrl to LR)
bdnzt 0,target	bc 8,0,target	Decrement CTR, branch if CTR nonzero AND condition true (bc relative)
bdnzta 0,target	bca 8,0,target	Decrement CTR, branch if CTR nonzero AND condition true (bca Absolute)
bdnztl 0,target	bcl 8,0,target	Decrement CTR, branch if CTR nonzero AND condition true (bcl relative)
bdnztle 0,target	bcla 8,0,target	Decrement CTR, branch if CTR nonzero AND condition true (bcla absolute)
bdnztlrl 0	bclrl 8,0	Decrement CTR, branch if CTR nonzero AND condition true (bclrl to LR)
bdz target	bc 18,0,target	Decrement CTR, branch if CTR zero (bc relative)
bdza target	bca 18,0,target	Decrement CTR, branch if CTR zero (bca Absolute)
bdzf 0,target	bc 2,0,target	Decrement CTR, branch if CTR zero AND condition false (bc relative)
bdzfa 0,target	bca 2,0,target	Decrement CTR, branch if CTR zero AND condition false (bca Absolute)
bdzfl 0,target	bcl 2,0,target	Decrement CTR, branch if CTR zero AND condition false (bcl relative)

Table A-25. Simplified Mnemonics (continued)

Simplified Mnemonic	Mnemonic	Instruction
bdzfla 0,target	bcla 2,0,target	Decrement CTR, branch if CTR zero AND condition false (bcla absolute)
bdzflr 0	bcctr 0,0	Decrement CTR, branch if CTR zero AND condition false (bclr to LR)
bdzl target	bcl 18,0,target	Decrement CTR, branch if CTR zero (bcl relative)
bdzla target	bcla 18,0,target	Decrement CTR, branch if CTR zero (bcla absolute)
bdzlr	bclr 18,0	Decrement CTR, branch if CTR zero (bclr to LR)
bdzlrl	bclrl 18,0	Decrement CTR, branch if CTR zero (bclrl to LR)
bdzt 0,target	bc 10,0,target	Decrement CTR, branch if CTR zero AND condition true (bc relative)
bdzta 0,target	bca 10,0,target	Decrement CTR, branch if CTR zero AND condition true (bca Absolute)
bdztl 0,target	bcl 10,0,target	Decrement CTR, branch if CTR zero AND condition true (bcl relative)
bdztla 0,target	bcla 10,0,target	Decrement CTR, branch if CTR zero AND condition true (bcla absolute)
bdztlr 0	bclr 10,0	Decrement CTR, branch if CTR nonzero AND condition true (bclr to LR)
bdztlr 0	bclr 10,0	Decrement CTR, branch if CTR zero AND condition true (bclr to LR)
bdztlrl 0	bdztlrl 0	Decrement CTR, branch if CTR zero AND condition true (bclrl to LR)
beq target	bc 12,2,target	Branch if equal (bc relative)
beqa target	bca 12,2,target	Branch if equal (bca absolute)
beqctr	bcctr 12,2	Branch if equal (bcctr to CTR)
beqctrl 0	bcctrl 12,2	Branch if equal (bcctrl to CTR)
beql target	bcl 12,2,target	Branch if equal (bcl relative)
beqla target	bcla 12,2,target	Branch if equal (bcla absolute)
beqlr	bclr 12,2	Branch if equal (bclr to LR)
beqlrl 0	bclrl 12,2	Branch if equal (bclrl to LR)
bf 0,target	bc 4,0,target	Branch if condition false (bc relative)
bfa 0,target	bca 4,0,target	Branch if condition false (bca Absolute)
bfctr 0	bcctr 4,0	Branch if condition false (bcctr to CTR)
bfctrl 0	bcctrl 4,0	Branch if condition false (bcctrl to CTR)
bfl 0,target	bcl 4,0,target	Branch if condition false (bcl relative)
bfla 0,target	bcla 4,0,target	Branch if condition false (bcla absolute)

Table A-25. Simplified Mnemonics (continued)

Simplified Mnemonic	Mnemonic	Instruction
bflr 0	bclr 4,0	Branch if condition false (bclr to LR)
bflrl 0	bclrl 4,0	Branch if condition false (bclrl to LR)
bflrl 0	bclrl 4,0	Decrement CTR, branch if CTR zero AND condition false (bclrl to LR)
bge target	bc 4,0,target	Branch if greater than or equal (bc relative)
bgea target	bca 4,0,target	Branch if greater than or equal (bca absolute)
bgectr	bcctr 4,0	Branch if greater than or equal (bcctr to CTR)
bgectrl 0	bcctrl 4,0	Branch if greater than or equal (bcctrl to CTR)
bgel target	bcl 4,0,target	Branch if greater than or equal (bcl relative)
bgela target	bcla 4,0,target	Branch if greater than or equal (bcla absolute)
bgelr	bclr 4,0	Branch if greater than or equal (bclr to LR)
bgelrl 0	bclrl 4,0	Branch if greater than or equal (bclrl to LR)
bgt target	bc 12,1,target	Branch if greater than (bc relative)
bgta target	bca 12,1,target	Branch if greater than (bca absolute)
bgctr	bcctr 12,1	Branch if greater than (bcctr to CTR)
bgctrl 0	bcctrl 12,1	Branch if greater than (bcctrl to CTR)
bgtl target	bcl 12,1,target	Branch if greater than (bcl relative)
bgtla target	bcla 12,1,target	Branch if greater than (bcla absolute)
bgtlr	bclr 12,1	Branch if greater than (bclr to LR)
bgtlrl 0	bclrl 12,1	Branch if greater than (bclrl to LR)
ble target	bc 4,1,target	Branch if less than or equal (bc relative)
blea target	bca 4,1,target	Branch if less than or equal (bca absolute)
blectr	bcctr 4,1	Branch if less than or equal (bcctr to CTR)
blectrl 0	bcctrl 4,1	Branch if less than or equal (bcctrl to CTR)
blel target	bcl 4,1,target	Branch if less than or equal (bcl relative)
blela target	bcla 4,1,target	Branch if less than or equal (bcla absolute)
blelr	bclr 4,1	Branch if less than or equal (bclr to LR)
blelrl 0	bclrl 4,1	Branch if less than or equal (bclrl to LR)
blr	bclr 20,0	Branch unconditionally (bclr to LR)
blrl	bclrl 20,0	Branch unconditionally (bclrl to LR)
blt target	bc 12,0,target	Branch if less than (bc relative)
blta target	bca 12,0,target	Branch if less than (bca absolute)
blctr	bcctr 12,0	Branch if less than (bcctr to CTR)
blctrl 0	bcctrl 12,0	Branch if less than (bcctrl to CTR)

Table A-25. Simplified Mnemonics (continued)

Simplified Mnemonic	Mnemonic	Instruction
bltl target	bcl 12,0,target	Branch if less than (bcl relative)
bltla target	bcla 12,0,target	Branch if less than (bcla absolute)
bltlr	bclr 12,0	Branch if less than (bclr to LR)
bltlrl 0	bclrl 12,0	Branch if less than (bclrl to LR)
bne target	bc 4,2,target	Branch if not equal (bc relative)
bnea target	bca 4,2,target	Branch if not equal (bca absolute)
bnecr	bcctr 4,2	Branch if not equal (bcctr to CTR)
bnecrl 0	bcctrl 4,2	Branch if not equal (bcctrl to CTR)
bnel target	bcl 4,2,target	Branch if not equal (bcl relative)
bnela target	bcla 4,2,target	Branch if not equal (bcla absolute)
bnelr	bclr 4,2	Branch if not equal (bclr to LR)
bnelrl 0	bclrl 4,2	Branch if not equal (bclrl to LR)
bng target	bc 4,1,target	Branch if not greater than (bc relative)
bnga target	bca 4,1,target	Branch if not greater than (bca absolute)
bngcr	bcctr 4,1	Branch if not greater than (bcctr to CTR)
bngcrl 0	bcctrl 4,1	Branch if not greater than (bcctrl to CTR)
bngl target	bcl 4,1,target	Branch if not greater than (bcl relative)
bngla target	bcla 4,1,target	Branch if not greater than (bcla absolute)
bnglr	bclr 4,1	Branch if not greater than (bclr to LR)
bnglrl 0	bclrl 4,1	Branch if not greater than (bclrl to LR)
bnl target	bc 4,0,target	Branch if not less than (bc relative)
bnla target	bca 4,0,target	Branch if not less than (bca absolute)
bnlcr	bcctr 4,0	Branch if not less than (bcctr to CTR)
bnlcrl 0	bcctrl 4,0	Branch if not less than (bcctrl to CTR)
bnll target	bcl 4,0,target	Branch if not less than (bcl relative)
bnlla target	bcla 4,0,target	Branch if not less than (bcla absolute)
bnllr	bclr 4,0	Branch if not less than (bclr to LR)
bnllrl 0	bclrl 4,0	Branch if not less than (bclrl to LR)
bns target	bc 4,3,target	Branch if not summary overflow (bc relative)
bnsa target	bca 4,3,target	Branch if not summary overflow (bca absolute)
bsncr	bcctr 4,3	Branch if not summary overflow (bcctr to CTR)
bsncrl 0	bcctrl 4,3	Branch if not summary overflow (bcctrl to CTR)
bnsll target	bcl 4,3,target	Branch if not summary overflow (bcl relative)

Table A-25. Simplified Mnemonics (continued)

Simplified Mnemonic	Mnemonic	Instruction
bnsia target	bcla 4,3,target	Branch if not summary overflow (bcla absolute)
bnsir	bclr 4,3	Branch if not summary overflow (bclr to LR)
bnsirl 0	bclrl 4,3	Branch if not summary overflow (bclrl to LR)
bnu target	bc 4,3,target	Branch if not unordered (bc relative)
bnu a target	bca 4,3,target	Branch if not unordered (bca absolute)
bnuctr	bcctr 4,3	Branch if not unordered (bcctr to CTR)
bnuctrl 0	bcctrl 4,3	Branch if not unordered (bcctrl to CTR)
bnul target	bcl 4,3,target	Branch if not unordered (bcl relative)
bnula target	bcla 4,3,target	Branch if not unordered (bcla absolute)
bnulr	bclr 4,3	Branch if not unordered (bclr to LR)
bnulrl 0	bclrl 4,3	Branch if not unordered (bclrl to LR)
bso target	bc 12,3,target	Branch if summary overflow (bc relative)
bsoa target	bca 12,3,target	Branch if summary overflow (bca absolute)
bsoctr	bcctr 12,3	Branch if summary overflow (bcctr to CTR)
bsoctrl 0	bcctrl 12,3	Branch if summary overflow (bcctrl to CTR)
bsol target	bcl 12,3,target	Branch if summary overflow (bcl relative)
bsola target	bcla 12,3,target	Branch if summary overflow (bcla absolute)
bsolr	bclr 12,3	Branch if summary overflow (bclr to LR)
bsolrl 0	bclrl 12,3	Branch if summary overflow (bclrl to LR)
bt 0,target	bc 12,0,target	Branch if condition true (bc relative)
bta 0,target	bca 12,0,target	Branch if condition true (bca Absolute)
btctr 0	bcctr 12,0	Branch if condition true (bcctr to CTR)
btctrl 0	bcctrl 12,0	Branch if condition true (bcctrl to CTR)
btl 0,target	bcl 12,0,target	Branch if condition true (bcl relative)
btla 0,target	bcla 12,0,target	Branch if condition true (bcla absolute)
btlr 0	bclr 12,0	Branch if condition true (bclr to LR)
btlrl 0	bclrl 12,0	Branch if condition true (bclrl to LR)
bun target	bc 12,3,target	Branch if unordered (bc relative)
buna target	bca 12,3,target	Branch if unordered (bca absolute)
bunctr	bcctr 12,3	Branch if unordered (bcctr to CTR)
bunctrl 0	bcctrl 12,3	Branch if unordered (bcctrl to CTR)
bunl target	bcl 12,3,target	Branch if unordered (bcl relative)
bunla target	bcla 12,3,target	Branch if unordered (bcla absolute)

Table A-25. Simplified Mnemonics (continued)

Simplified Mnemonic	Mnemonic	Instruction
bunlr	bclr 12,3	Branch if unordered (bclr to LR)
bunlrl 0	bclrl 12,3	Branch if unordered (bclrl to LR)
clrlslwi rA,rS,b,n ($n \leq b \leq 31$)	rlwinm rA,rS,n,b - n,31 - n	Clear left and shift left immediate
clrlwi rA,rS,n ($n < 32$)	rlwinm rA,rS,0,n,31	Clear left immediate
clrrwi rA,rS,n ($n < 32$)	rlwinm rA,rS,0,0,31 - n	Clear right immediate
cmplw crfD,rA,rB	cmpl crfD,0,rA,rB	Compare Logical Word
cmplwi crfD,rA,UIMM	cmpli crfD,0,rA,UIMM	Compare Logical Word Immediate
cmpw crfD,rA,rB	cmp crfD,0,rA,rB	Compare Word
cmpwi crfD,rA,SIMM	cmpi crfD,0,rA,SIMM	Compare Word Immediate
crclr bx	crxor bx,bx,bx	Condition register clear
crmove bx,by	cror bx,by,by	Condition register move
crnot bx,by	crnor bx,by,by	Condition register not
crset bx	creqv bx,bx,bx	Condition register set
extlwi rA,rS,n,b ($n > 0$)	rlwinm rA,rS,b,0,n - 1	Extract and left justify immediate
extrwi rA,rS,n,b ($n > 0$)	rlwinm rA,rS,b + n, 32 - n,31	Extract and right justify immediate
inslwi rA,rS,n,b ($n > 0$)	rlwimi rA,rS,32 - b,b,(b + n) - 1	Insert from left immediate
insrwi rA,rS,n,b ($n > 0$)	rlwimi rA,rS,32 - (b + n),b,(b + n) - 1	Insert from right immediate
la rD,d(rA)	addi rD,rA,d	Load address
li rD,value	addi rD,0,value	Load immediate
lis rD,value	addis rD,0,value	Load immediate signed
mfcsrr0 rD	mfspr 58, rD	Move from SPR: Critical save/restore register 0
mfcsrr1 rD	mfspr 59, rD	Move from SPR: Critical save/restore register 1
mfctr rD	mfspr 9, rD	Move from SPR: Count register
mfdac1 rD	mfspr 316, rD	Move from SPR: Data address compare 1
mfdac2 rD	mfspr 317, rD	Move from SPR: Data address compare 2
mfdbcr0 rD	mfspr 308, rD	Move from SPR: Debug control register 0
mfdbcr1 rD	mfspr 309, rD	Move from SPR: Debug control register 1
mfdbcr2 rD	mfspr 310, rD	Move from SPR: Debug control register 2
mfdbsr rD	mfspr 304, rD	Move from SPR: Debug status register
mfdear rD	mfspr 61, rD	Move from SPR: Data exception address register
mfdec rD	mfspr 22, rD	Move from SPR: Decrementer
mfdvc1 rD	mfspr 318, rD	Move from SPR: Data value compare 1
mfdvc2 rD	mfspr 319, rD	Move from SPR: Data value compare 2

Table A-25. Simplified Mnemonics (continued)

Simplified Mnemonic	Mnemonic	Instruction
mfesr rD	mf spr 62, rD	Move from SPR: Exception syndrome register
mfiacl rD	mf spr 312, rD	Move from SPR: Instruction address compare 1
mfiacl2 rD	mf spr 313, rD	Move from SPR: Instruction address compare 2
mfiacl3 rD	mf spr 314, rD	Move from SPR: Instruction address compare 3
mfiacl4 rD	mf spr 315, rD	Move from SPR: Instruction address compare 4
mfivor0 rD	mf spr 400, rD	Move from SPR: Interrupt vector offset register 0
mfivor1 rD	mf spr 401, rD	Move from SPR: Interrupt vector offset register 1
mfivor10 rD	mf spr rD, 410	Move from SPR: Interrupt vector offset register 10
mfivor11 rD	mf spr rD, 411	Move from SPR: Interrupt vector offset register 11
mfivor12 rD	mf spr rD, 412	Move from SPR: Interrupt vector offset register 12
mfivor13 rD	mf spr rD, 413	Move from SPR: Interrupt vector offset register 13
mfivor14 rD	mf spr rD, 414	Move from SPR: Interrupt vector offset register 14
mfivor15 rD	mf spr rD, 415	Move from SPR: Interrupt vector offset register 15
mfivor2 rD	mf spr 402, rD	Move from SPR: Interrupt vector offset register 2
mfivor3 rD	mf spr 403, rD	Move from SPR: Interrupt vector offset register 3
mfivor4 rD	mf spr 404, rD	Move from SPR: Interrupt vector offset register 4
mfivor5 rD	mf spr 405, rD	Move from SPR: Interrupt vector offset register 5
mfivor6 rD	mf spr 406, rD	Move from SPR: Interrupt vector offset register 6
mfivor7 rD	mf spr 407, rD	Move from SPR: Interrupt vector offset register 7
mfivor8 rD	mf spr 408, rD	Move from SPR: Interrupt vector offset register 8
mfivor9 rD	mf spr 409, rD	Move from SPR: Interrupt vector offset register 9
mfivpr rD	mf spr 63, rD	Move from SPR: Interrupt vector prefix register
mflr rD	mf spr rD, 8	Move from SPR: Link register
mfpid rD	mf spr rD, 48	Move from SPR: Process ID register
mfpir rD	mf spr rD, 286	Move from SPR: Processor ID register
mfpvr rD	mf spr rD, 287	Move from SPR: Processor version register
mf sprg0 rD	mf spr rD, 272	Move from SPR: SPR general 0
mf sprg1 rD	mf spr rD, 273	Move from SPR: SPR general 1
mf sprg2 rD	mf spr rD, 274	Move from SPR: SPR general 2

Table A-25. Simplified Mnemonics (continued)

Simplified Mnemonic	Mnemonic	Instruction
mfsprg3 rD	mfspr rD, 259	Move from SPR: SPR general 3 (user/supervisor)
mfsprg3 rD	mfspr rD, 275	Move from SPR: SPR general 3 (supervisor only)
mfsprg4 rD	mfspr rD, 260	Move from SPR: SPR general 4 (user/supervisor)
mfsprg4 rD	mfspr rD, 276	Move from SPR: SPR general 4 (supervisor only)
mfsprg5 rD	mfspr rD, 261	Move from SPR: SPR general 5 (user/supervisor)
mfsprg5 rD	mfspr rD, 277	Move from SPR: SPR general 5 (supervisor only)
mfsprg6 rD	mfspr rD, 262	Move from SPR: SPR general 6 (user/supervisor)
mfsprg6 rD	mfspr rD, 278	Move from SPR: SPR general 6 (supervisor only)
mfsprg7 rD	mfspr rD, 263	Move from SPR: SPR general 7 (user/supervisor)
mfsprg7 rD	mfspr rD, 279	Move from SPR: SPR general 7 (supervisor only)
mfsrc0 rD	mfspr rD, 26	Move from SPR: Save/restore register 0
mfsrc1 rD	mfspr rD, 27	Move from SPR: Save/restore register 1
mftbl rD	mfspr rD, 268	Move from SPR: Time base lower
mftbu rD	mfspr rD, 269	Move from SPR: Time base upper
mftcr rD	mfspr rD, 340	Move from SPR: Timer control register
mftsr rD	mfspr rD, 336	Move from SPR: Timer status register
mfusprg0 rD	mfspr rD, 256	Move from SPR: User SPR general 0
mfxc rD	mfspr rD, 1	Move from SPR: Integer exception register
mr rA,rS	or rA,rS,rS	Move register
mtbbear rS	mtspr 513, rS	Move to SPR: Branch buffer entry address register
mtbbtar rS	mtspr 514, rS	Move to SPR: Branch buffer target address register
mtbucsr rS	mtspr 1013, rS	Move to SPR: Branch unit control and status register 0
mtcr rS	mtcrf 0xFF,rS	Move to condition register
mtcsrr0 rS	mtspr 58, rS	Move to SPR: Critical save/restore register 0
mtcsrr1 rS	mtspr 59, rS	Move to SPR: Critical save/restore register 1

Table A-25. Simplified Mnemonics (continued)

Simplified Mnemonic	Mnemonic	Instruction
mtctr rS	mtspr 9, rS	Move to SPR: Count register
mtdac1 rS	mtspr 316, rS	Move to SPR: Data address compare 1
mtdac2 rS	mtspr 317, rS	Move to SPR: Data address compare 2
mtdbcr0 rS	mtspr 308, rS	Move to SPR: Debug control register 0
mtdbcr1 rS	mtspr 309, rS	Move to SPR: Debug control register 1
mtdbcr2 rS	mtspr 310, rS	Move to SPR: Debug control register 2
mtdbsr rS	mtspr 304, rS	Move to SPR: Debug status register
mtdear rS	mtspr 61, rS	Move to SPR: Data exception address register
mtdec rS	mtspr 22, rS	Move to SPR: Decrementer
mtdecar rS	mtspr 54, rS	Move to SPR: Decrementer auto-reload
mtdvc1 rS	mtspr 318, rS	Move to SPR: Data value compare 1
mtdvc2 rS	mtspr 319, rS	Move to SPR: Data value compare 2
mtesr rS	mtspr 62, rS	Move to SPR: Exception syndrome register
mthid0 rS	mtspr 1008, rS	Move to SPR: Hardware implementation dependent reg 0
mthid1 rS	mtspr 1009, rS	Move to SPR: Hardware implementation dependent reg 1
mtiac1 rS	mtspr 312, rS	Move to SPR: Instruction address compare 1
mtiac2 rS	mtspr 313, rS	Move to SPR: Instruction address compare 2
mtiac3 rS	mtspr 314, rS	Move to SPR: Instruction address compare 3
mtiac4 rS	mtspr 315, rS	Move to SPR: Instruction address compare 4
mtiarr rS	mtspr 518, rS	Move to SPR: Instruction address resume register
mtivor0 rS	mtspr 400, rS	Move to SPR: Interrupt vector offset register 0
mtivor1 rS	mtspr 401, rS	Move to SPR: Interrupt vector offset register 1
mtivor10 rS	mtspr 410, rS	Move to SPR: Interrupt vector offset register 10
mtivor11 rS	mtspr 411, rS	Move to SPR: Interrupt vector offset register 11
mtivor12 rS	mtspr 412, rS	Move to SPR: Interrupt vector offset register 12
mtivor13 rS	mtspr 413, rS	Move to SPR: Interrupt vector offset register 13
mtivor14 rS	mtspr 414, rS	Move to SPR: Interrupt vector offset register 14
mtivor15 rS	mtspr 415, rS	Move to SPR: Interrupt vector offset register 15
mtivor2 rS	mtspr 402, rS	Move to SPR: Interrupt vector offset register 2
mtivor3 rS	mtspr 403, rS	Move to SPR: Interrupt vector offset register 3
mtivor32 rS	mtspr 528, rS	Move to SPR: Interrupt vector offset register 32 (SPU interrupt)

Table A-25. Simplified Mnemonics (continued)

Simplified Mnemonic	Mnemonic	Instruction
mtivor33 rS	mtspr 529, rS	Move to SPR: Interrupt vector offset register 33 (Vector floating-point data exception)
mtivor34 rS	mtspr 530, rS	Move to SPR: Interrupt vector offset register 34 (Vector floating-point round exception)
mtivor35 rS	mtspr 531, rS	Move to SPR: Interrupt vector offset register 35 (performance monitor)
mtivor4 rS	mtspr 404, rS	Move to SPR: Interrupt vector offset register 4
mtivor5 rS	mtspr 405, rS	Move to SPR: Interrupt vector offset register 5
mtivor6 rS	mtspr 406, rS	Move to SPR: Interrupt vector offset register 6
mtivor7 rS	mtspr 407, rS	Move to SPR: Interrupt vector offset register 7
mtivor8 rS	mtspr 408, rS	Move to SPR: Interrupt vector offset register 8
mtivor9 rS	mtspr 409, rS	Move to SPR: Interrupt vector offset register 9
mtivpr rS	mtspr 63, rS	Move to SPR: Interrupt vector prefix register
mtl1csr0 rS	mtspr 1010, rS	Move to SPR: L1 cache control and status register 0
mtl1csr1 rS	mtspr 1011, rS	Move to SPR: L1 cache control and status register 1
mtlr rS	mtspr 8, rS	Move to SPR: Link register
mtmas0 rS	mtspr 624, rS	Move to SPR: MMU assist register 0
mtmas1 rS	mtspr 625, rS	Move to SPR: MMU assist register 1
mtmas2 rS	mtspr 626, rS	Move to SPR: MMU assist register 2
mtmas3 rS	mtspr 627, rS	Move to SPR: MMU assist register 3
mtmas4 rS	mtspr 628, rS	Move to SPR: MMU assist register 4
mtmas6 rS	mtspr 630, rS	Move to SPR: MMU assist register 6
mtmcsr rS	mtspr 572, rS	Move to SPR:
mtmcsrr0 rS	mtspr 570, rS	Move to SPR: Machine-check save/restore register 0
mtmcsrr1 rS	mtspr 571, rS	Move to SPR: Machine-check save/restore register 1
mtmmucsr0 rS	mtspr 1012, rS	Move to SPR: MMU control and status register 0
mtnpidr rS	mtspr 517, rS	Move to SPR: Nexus processor ID register
mtpid rS	mtspr 48, rS	Move to SPR: Process ID register
mtpid rS	mtspr 48, rS	Move to SPR: Process ID register 0. Book E defines only this PID register and refers to as PID rather than PID0.
mtpid1 rS	mtspr 633, rS	Move to SPR: Process ID register 1
mtpid2 rS	mtspr 634, rS	Move to SPR: Process ID register 2

Table A-25. Simplified Mnemonics (continued)

Simplified Mnemonic	Mnemonic	Instruction
mtsprg0 rS	mtspr 272, rS	Move to SPR: SPR general 0
mtsprg1 rS	mtspr 273, rS	Move to SPR: SPR general 1
mtsprg2 rS	mtspr 274, rS	Move to SPR: SPR general 2
mtsprg3 rS	mtspr 275, rS	Move to SPR: SPR general 3
mtsprg4 rS	mtspr 276, rS	Move to SPR: SPR general 4
mtsprg5 rS	mtspr 277, rS	Move to SPR: SPR general 5
mtsprg6 rS	mtspr 278, rS	Move to SPR: SPR general 6
mtsprg7 rS	mtspr 279, rS	Move to SPR: SPR general 7
mtsrr0 rS	mtspr 26, rS	Move to SPR: Save/restore register 0
mtsrr1 rS	mtspr 27, rS	Move to SPR: Save/restore register 1
mtspfcscr rS	mtspr 512, rS	Move to SPR: Signal processing and embedded floating-point status and control register
mttbl rS	mtspr 284, rS	Move to SPR: Time base lower (supervisor only)
mttbu rS	mtspr 285, rS	Move to SPR: Time base upper (supervisor only)
mttcr rS	mtspr 340, rS	Move to SPR: Timer control register
mttsr rS	mtspr 336, rS	Move to SPR: Timer status register
mtusprg0 rS	mtspr 256, rS	Move to SPR: User SPR general 0
mtxer rS	mtspr 1, rS	Move to SPR: Integer exception register
nop	ori 0,0,0	No-op
not rA,rS	nor rA,rS,rS	Complement register
rotlw rA,rS,rB	rlwnm rA,rS,rB,0,31	Rotate left
rotlwi rA,rS,n	rlwinm rA,rS,n,0,31	Rotate left immediate
rotrwi rA,rS,n	rlwinm rA,rS,32 – n,0,31	Rotate right immediate
slwi rA,rS,n ($n < 32$)	rlwinm rA,rS,n,0,31 – n	Shift left immediate
srwi rA,rS,n ($n < 32$)	rlwinm rA,rS,32 – n,n,31	Shift right immediate
sub rD,rA,rB	subf rD,rB,rA	Subtract from
subc rD,rA,rB	subfc rD,rB,rA	Subtract from Carrying
subi rD,rA,value	addi rD,rA,–value	Subtract immediate
subic rD,rA,value	addic rD,rA,–value	Subtract immediate Carrying
subic. rD,rA,value	addic. rD,rA,–value	Subtract immediate Carrying
subis rD,rA,value	addis rD,rA,–value	Subtract immediate signed
tweqi	tweq	Trap if equal
twgei	twge	Trap if greater than or equal
twgti	twgt	Trap if greater than

Table A-25. Simplified Mnemonics (continued)

Simplified Mnemonic	Mnemonic	Instruction
twlei	twle	Trap if less than or equal
twlgei	twlge	Trap if logically greater than or equal
twlgti	twlgt	Trap if logically greater than
twllei	twlle	Trap if logically less than or equal
twllti	twllt	Trap if logically less than
twlngi	twlng	Trap if logically not greater than
twlnli	twlnl	Trap if logically not less than
twlti	twlt	Trap if less than
twnei	twne	Trap if not equal
twngi	twng	Trap if not greater than
twnli	twnl	Trap if not less than

Appendix B

Programming Examples

This appendix gives examples of how memory synchronization instructions can be used to emulate various synchronization primitives and to provide more complex forms of synchronization. It also describes multiple precision shifts.

B.1 Synchronization

Examples in this appendix have a common form. After possible initialization, a conditional sequence begins with a load and reserve instruction that may be followed by memory accesses and computations that include neither a load and reserve nor a store conditional. The sequence ends with a store conditional with the same target address as the initial load and reserve. In most of the examples, failure of the store conditional causes a branch back to the load and reserve for a repeated attempt. On the assumption that contention is low, the conditional branch in the examples is optimized for the case in which the store conditional succeeds, by setting the branch-prediction bit appropriately. These examples focus on techniques for the correct modification of shared storage locations: see note 4 in Section B.1.4, “Notes,” for a discussion of how the retry strategy can affect performance.

Load and reserve and store conditional instructions depend on the coherence mechanism of the system. Stores to a given location are coherent if they are serialized in some order, and no processor is able to observe a subset of those stores as occurring in a conflicting order. See Section 11.1.3.1, “Memory Access Ordering,” for details.

Each load operation, whether ordinary or load and reserve, returns a value that has a well-defined source. The source can be the store or store conditional instruction that wrote the value, an operation by some other mechanism that accesses storage (for example, an I/O device), or the initial state of storage.

The function of an atomic read/modify/write operation is to read a location and write its next value, possibly as a function of its current value, all as a single atomic operation. We assume that locations accessed by read/modify/write operations are accessed coherently, so the concept of a value being the next in the sequence of values for a location is well defined. The conditional sequence, as defined above, provides the effect of an atomic read/modify/write operation, but not with a single atomic instruction. Let *addr* be the location that is the common target of the load and reserve and store conditional instructions.

Synchronization

Then the guarantee the architecture makes for the successful execution of the conditional sequence is that no store into `addr` by another processor or mechanism has intervened between the source of the load and reserve and the store conditional.

For each of these examples, it is assumed that a similar sequence of instructions is used by all processes requiring synchronization on the accessed data.

NOTE

Because memory synchronization instructions have implementation dependencies (for example, the granularity at which reservations are managed), they must be used with care. The operating system should provide system library programs that use these instructions to implement the high-level synchronization functions (such as, test and set or compare and swap) needed by application programs. Application programs should use these library programs, rather than use storage synchronization instructions directly.

B.1.1 Synchronization Primitives

The following examples show how the **lwarx** and **stwcx.** instructions can be used to implement various synchronization primitives.

The sequences used to emulate the various primitives consist primarily of a loop using **lwarx** and **stwcx.** No additional synchronization is necessary, because the **stwcx.** will fail, clearing EQ, if the word loaded by **lwarx** has changed before the **stwcx.** is executed: see Section 3.4.1.8, “Atomic Update Primitives Using **lwarx** and **stwcx.**,” for details.

B.1.1.1 Fetch and No-op

The fetch and no-op primitive atomically loads the current value in a word in storage.

In this example it is assumed that the address of the word to be loaded is in GPR3 and the data loaded are returned in GPR4.

```
loop:  lwarx    r4,0,r3          #load and reserve
       stwcx.  r4,0,r3          #store old value if still reserved
       bc      4,2,loop         #loop if lost reservation
```

If the **stwcx.** succeeds, it stores to the target location the same value that was loaded by the preceding **lwarx**. While the store is redundant with respect to the value in the location, its success ensures that the value loaded by the **lwarx** was the current value, that is, that the source of the value loaded by the **lwarx** was the last store to the location that preceded the **stwcx.** in the coherence order for the location.

B.1.1.2 Fetch and Store

The fetch and store primitive atomically loads and replaces a word in storage. In this example it is assumed that the address of the word to be loaded and replaced is in GPR3, the new value is in GPR4, and the old value is returned in GPR5.

```
loop:  lwarx    r5,0,r3      #load and reserve
       stwcx.   r4,0,r3      #store new value if still reserved
       bc      4,2,loop     #loop if lost reservation
```

B.1.1.3 Fetch and Add

The fetch and add primitive atomically increments a word in storage. In this example it is assumed that the address of the word to be incremented is in GPR3, the increment is in GPR4, and the old value is returned in GPR5.

```
loop:  lwarx    r5,0,r3      #load and reserve
       add     r0,r4,r5      #increment word
       stwcx.   r0,0,r3      #store new value if still reserved
       bc      4,2,loop     #loop if lost reservation
```

B.1.1.4 Fetch and AND

The Fetch and AND primitive atomically ANDs a value into a word in storage.

In this example it is assumed that the address of the word to be ANDed is in GPR3, the value to AND into it is in GPR4, and the old value is returned in GPR5.

```
loop:  lwarx    r5,0,r3      #load and reserve
       and     r0,r4,r5      #AND word
       stwcx.   r0,0,r3      #store new value if still reserved
       bc      4,2,loop     #loop if lost reservation
```

This sequence can be changed to perform another Boolean operation atomically on a word in memory by changing the **and** to the desired Boolean instruction (**or**, **xor**, etc.).

B.1.1.5 Test and Set

This version of the test and set primitive atomically loads a word from memory, sets the word in memory to a nonzero value if the value loaded is zero, and sets the EQ bit of CR Field 0 to indicate whether the value loaded is zero.

In this example it is assumed that the address of the word to be tested is in GPR3, the new value (nonzero) is in GPR4, and the old value is returned in GPR5.

```
loop:  lwarx    r5,0,r3      #load and reserve
       cmpwi   r5,0         #done if word
       bc      4,2,done     # not equal to 0
       stwcx.   r4,0,r3      #try to store non-0
       bc      4,2,loop     #loop if lost reservation
done:
```

B.1.1.6 Compare and Swap

The compare and swap primitive atomically compares a value in a register with a word in memory, if they are equal stores the value from a second register into the word in memory, if they are unequal loads the word from memory into the first register, and sets CR0[EQ] to indicate the result of the comparison.

In this example it is assumed that the address of the word to be tested is in GPR3, the comparand is in GPR4 and the old value is returned there, and the new value is in GPR5.

```

loop:  lwarx    r6,0,r3          #load and reserve
        cmpw    r4,r6          #1st 2 operands equal?
        bc      4,2,exit       #skip if not
        stwcx.  r5,0,r3        #store new value if still reserved
        bc      4,2,loop       #loop if lost reservation
exit:   or      r4,r6,r6        #return value from memory

```

Notes:

1. The semantics given for compare and swap above are based on those of the IBM System/370 compare and swap instruction. Other architectures may define a compare and swap instruction differently.
2. Compare and swap is shown primarily for pedagogical reasons. It is useful on machines that lack the better synchronization facilities provided by **lwarx** and **stwcx..** A major weakness of a System/370-style compare and swap instruction is that, although the instruction itself is atomic, it checks only that the old and current values of the word being tested are equal, with the result that programs that use such a compare and swap to control a shared resource can err if the word has been modified and the old value subsequently restored. The sequence shown above has the same weakness.
3. In some applications the second **bc** and/or the **or** can be omitted. The **bc** is needed only if the application requires that if CR0[EQ] on exit indicates not equal then GPR4 and GPR6 are not equal. The **or** is needed only if the application requires that if the comparands are not equal then the word from memory is loaded into the register with which it was compared (rather than into a third register). If any of these instructions is omitted, the resulting compare and swap does not obey System/370 semantics.

B.1.2 Lock Acquisition and Release

This example gives an algorithm for locking that demonstrates the use of synchronization with an atomic read/modify/write operation. A shared memory location, the address of which is an argument of the lock and unlock procedures, given by GPR3, is used as a lock, to control access to some shared resource such as a shared data structure. The lock is open when its value is 0 and closed (locked) when its value is 1. Before accessing the shared resource the program executes the lock procedure, which sets the lock by changing its value

from 0 to 1. To do this, the lock procedure calls `test_and_set`, which executes the code sequence shown in the test and set example of Section B.1.1, “Synchronization Primitives,” thereby atomically loading the old value of the lock, writing to the lock the new value (1) given in GPR4, returning the old value in GPR5 (not used below), and setting the EQ bit of CR Field 0 according to whether the value loaded is 0. The lock procedure repeats the `test_and_set` until it succeeds in changing the value of the lock from 0 to 1.

Because the shared resource must not be accessed until the lock has been set, the lock procedure contains an **isync** after the **bc** that checks for the success of `test_and_set`. The **isync** delays all subsequent instructions until all preceding instructions have completed.

```
lock:  mfspr    r6,LR           #save Link Register
      addi    r4,r0,1         #obtain lock:
loop:  bl     test_and_set     # test-and-set
      bc     4,2,loop         # retry til old = 0
# Delay subsequent instructions til prior instructions finish
      isync
      mtspr   LR,r6           #restore Link Register
      blr                                #return
```

The unlock procedure stores a 0 to the lock location. Most applications that use locking require, for correctness, that if the access to the shared resource includes stores, the program must execute an **msync** before releasing the lock. The **msync** ensures that the program's modifications are performed with respect to other processors before the store that releases the lock is performed with respect to those processors. In this example, the unlock procedure begins with an **msync** for this purpose.

```
unlock: msync                #order prior stores
      addi    r1,r0,0         #before lock release
      stw     r1,0(r3)        #store 0 to lock location
      blr                                #return
```

B.1.3 List Insertion

This example shows how **lwarx** and **stwcx** can be used to implement simple insertion into a singly linked list. (Complicated list insertion, in which multiple values must be changed atomically, or in which the correct order of insertion depends on the contents of the elements, cannot be implemented in the manner shown below and requires a more complicated strategy such as using locks.)

The next element pointer from the list element after which the new element is to be inserted, here called the parent element, is stored into the new element, so that the new element points to the next element in the list: this store is performed unconditionally. Then the address of the new element is conditionally stored into the parent element, thereby adding the new element to the list.

In this example it is assumed that the address of the parent element is in GPR3, the address of the new element is in GPR4, and the next element pointer is at offset 0 from the start of

Synchronization

the element. It is also assumed that the next element pointer of each list element is in a reservation granule separate from that of the next element pointer of all other list elements: see Section 3.4.1.8, “Atomic Update Primitives Using `lwarx` and `stwcx.`”

```
loop:  lwarx    r2,0,r3          #get next pointer
      stw      r2,0(r4)         #store in new element
      msync    #order stw before stwcx.(can omit if not MP)
      stwcx.   r4,0,r3          #add new element to list
      bc      4,2,loop          #loop if stwcx. failed
```

In the preceding example, if two list elements have next element pointers in the same reservation granule then, in a multiprocessor, livelock can occur. (Livelock is a state in which processors interact in a way such that no processor makes progress.)

If it is not possible to allocate list elements such that each element's next element pointer is in a different reservation granule, then livelock can be avoided by using the following, more complicated, sequence.

```
      lwz      r2,0(r3)         #get next pointer
loop1: or      r5,r2,r2          #keep a copy
      stw      r2,0(r4)         #store in new element
      msync    #order stw before stwcx.
loop2: lwarx    r2,0,r3          #get it again
      cmpw     r2,r5            #loop if changed (someone
      bc      4,2,loop1         # else progressed)
      stwcx.   r4,0,r3          #add new element to list
      bc      4,2,loop          #loop if failed
```

B.1.4 Notes

1. In general, **`lwarx`** and **`stwcx.`** should be paired, with the same effective address used for both. The only exception is that an unpaired **`stwcx.`** to any (scratch) effective address can be used to clear any reservation held by the processor.
2. It is acceptable to execute a **`lwarx`** for which no **`stwcx.`** is executed. For example, this occurs in the test and set sequence shown above if the value loaded is not zero.
3. To increase the likelihood that forward progress is made, it is important that looping on **`lwarx/stwcx.`** pairs be minimized. For example, in the sequence shown above for test and set, this is achieved by testing the old value before attempting the store: were the order reversed, more **`stwcx.`** instructions might be executed, and reservations might more often be lost between the **`lwarx`** and the **`stwcx.`**.
4. The manner in which **`lwarx`** and **`stwcx.`** are communicated to other processors and mechanisms, and between levels of the memory subsystem within a given processor is implementation-dependent (see Section 3.4.1.8, “Atomic Update Primitives Using `lwarx` and `stwcx.`”). In some implementations performance may be improved by minimizing looping on a **`lwarx`** instruction that fails to return a desired value. For example, in the test and set example shown above, if the programmer wishes to stay in the loop until the word loaded is zero, he could

change the `bne- $+12` to `bne- loop`. However, in some implementations better performance may be obtained by using an ordinary load instruction to do the initial checking of the value, as follows.

```
loop:  lwz      r5,0(r3)          #load the word
      cmpi    cr0,0,r5,0        #loop back if word
      bc      4,2,loop          # not equal to 0
      lwarx   r5,0,r3           #try again, reserving
      cmpi    cr0,0,r5,0        # (likely to succeed)
      bc      4,2,loop          #
      stwcx.  r4,0,r3           #try to store non-0
      bc      4,2,loop          #loop if lost reservation
```

5. In a multiprocessor, livelock is possible if a loop containing a **lwarx/stwcx**. pair also contains an ordinary store instruction for which any byte of the affected memory area is in the reservation granule: see Section 3.4.1.8, “Atomic Update Primitives Using lwarx and stwcx.” For example, the first code sequence shown in Section B.1.3, “List Insertion,” can cause livelock if two list elements have next element pointers in the same reservation granule.

B.2 Multiple-Precision Shifts

This section gives examples of how multiple-precision shifts can be programmed.

A multiple-precision shift is defined to be a shift of an N-doubleword quantity (64-bit implementations) or an N-word quantity (32-bit implementations), where $N > 1$. The quantity to be shifted is contained in N registers. The shift amount is specified either by an immediate value in the instruction or by a value in a register.

The examples shown below distinguish between the cases $N=2$ and $N>2$. If $N=2$, the shift amount may be in the range 0 through 127 (64-bit implementations) or 0 through 63 (32-bit implementations), which are the maximum ranges supported by the *Shift* instructions used. However if $N>2$, the shift amount must be in the range 0 through 63 (64-bit implementations) or 0 through 31 (32-bit implementations), in order for the examples to yield the desired result. The specific instance shown for $N>2$ is $N=3$: extending those code sequences to larger N is straightforward, as is reducing them to the case $N=2$ when the more stringent restriction on shift amount is met. For shifts with immediate shift amounts only the case $N=3$ is shown, because the more stringent restriction on shift amount is always met.

In the examples it is assumed that GPRs 2 and 3 (and 4) contain the quantity to be shifted, and that the result is to be placed into the same registers, except for the immediate left shifts in 64-bit implementations, for which the result is placed into GPRs 3, 4, and 5. In all cases, for both input and result, the lowest-numbered register contains the highest-order part of the data and highest-numbered register contains the lowest-order part. For non-immediate shifts, the shift amount is assumed to be in GPR6. For immediate shifts, the shift amount is assumed to be greater than 0. GPRs 0 and 31 are used as scratch registers.

Multiple-Precision Shifts

For $N > 2$, the number of instructions required is $2N-1$ (immediate shifts) or $3N-1$ (non-immediate shifts).

Table B-1. Shifts in 32- and 64-Bit Implementations

64-Bit Implementations	32-Bit Implementations
Shift Left Immediate, N=3 (shift amount < 64) rldicr r5, r4, sh, 63-sh rldimi r4, r3, 0, sh rldicl r4, r4, sh, 0 rldimi r3, r2, 0, sh rldicl r3, r3, sh, 0	Shift Left Immediate, N=3 (shift amount < 32) rlwinm r2, r2, sh, 0, 31-sh rlwimi r2, r3, sh, 32-sh, 31 rlwinm r3, r3, sh, 0, 31-sh rlwimi r3, r4, sh, 32-sh, 31 rlwinm r4, r4, sh, 0, 31-sh
Shift Left, N=2 (shift amount < 128) subfic r31, r6, 64 sld r2, r2, r6 srd r0, r3, r31 or r2, r2, r0 addi r31, r6, -64 sld r0, r3, r31 or r2, r2, r0 sld r3, r3, r6	Shift Left, N=2 (shift amount < 64) subfic r31, r6, 32 slw r2, r2, r6 srw r0, r3, r31 or r2, r2, r0 addi r31, r6, -32 slw r0, r3, r31 or r2, r2, r0 slw r3, r3, r6
Shift Left, N=3 (shift amount < 64) subfic r31, r6, 64 sld r2, r2, r6 srd r0, r3, r31 or r2, r2, r0 sld r3, r3, r6 srd r0, r4, r31 or r3, r3, r0 sld r4, r4, r6	Shift Left, N=3 (shift amount < 32) subfic r31, r6, 32 slw r2, r2, r6 srw r0, r3, r31 or r2, r2, r0 slw r3, r3, r6 srw r0, r4, r31 or r3, r3, r0 slw r4, r4, r6
Shift Right Immediate, N=3 (shift amount < 64) rldimi r4, r3, 0, 64-sh rldicl r4, r4, 64-sh, 0 rldimi r3, r2, 0, 64-sh rldicl r3, r3, 64-sh, 0 rldicl r2, r2, 64-sh, sh	Shift Right Immediate, N=3 (shift amount < 32) rlwinm r4, r4, 32-sh, sh, 31 rlwimi r4, r3, 32-sh, 0, sh-1 rlwinm r3, r3, 32-sh, sh, 31 rlwimi r3, r2, 32-sh, 0, sh-1 rlwinm r2, r2, 32-sh, sh, 31
Shift Right, N=2 (shift amount < 128) subfic r31, r6, 64 srd r3, r3, r6 sld r0, r2, r31 or r3, r3, r0 addi r31, r6, -64 srd r0, r2, r31 or r3, r3, r0 srd r2, r2, r6	Shift Right, N=2 (shift amount < 64) subfic r31, r6, 32 srw r3, r3, r6 slw r0, r2, r31 or r3, r3, r0 addi r31, r6, -32 srw r0, r2, r31 or r3, r3, r0 srw r2, r2, r6
Shift Right, N=3 (shift amount < 64) subfic r31, r6, 64 srd r4, r4, r6 sld r0, r3, r31 or r4, r4, r0 srd r3, r3, r6 sld r0, r2, r31 or r3, r3, r0 srd r2, r2, r6	Shift Right, N=3 (shift amount < 32) subfic r31, r6, 32 srw r4, r4, r6 slw r0, r3, r31 or r4, r4, r0 srw r3, r3, r6 slw r0, r2, r31 or r3, r3, r0 srw r2, r2, r6

Table B-1. Shifts in 32- and 64-Bit Implementations (continued)

64-Bit Implementations	32-Bit Implementations
Shift Right Algebraic Immediate, N=3 (shift amt < 64) rldimi r4, r3, 0, 64-sh rldicl r4, r4, 64-sh, 0 rldimi r3, r2, 0, 64-sh rldicl r3, r3, 64-sh, 0 sradi r2, r2, sh	Shift Right Algebraic Immediate, N=3 (shift amt < 32) rlwinm r4, r4, 32-sh, sh, 31 rlwimi r4, r3, 32-sh, 0, sh-1 rlwinm r3, r3, 32-sh, sh, 31 rlwimi r3, r2, 32-sh, 0, sh-1 srawi r2, r2, sh
Shift Right Algebraic, N=2 (shift amount < 128) subfic r31, r6, 64 srd r3, r3, r6 sld r0, r2, r31 or r3, r3, r0 addic. r31, r6, -64 srad r0, r2, r31 bc 4, 1, \$+8 ori r3, r0, 0 srad r2, r2, r6	Shift Right Algebraic, N=2 (shift amount < 64) subfic r31, r6, 32 srw r3, r3, r6 slw r0, r2, r31 or r3, r3, r0 addic. r31, r6, -32 sraw r0, r2, r31 bc 4, 1, \$+8 ori r3, r0, 0 sraw r2, r2, r6
Shift Right Algebraic, N=3 (shift amount < 64) subfic r31, r6, 64 srd r4, r4, r6 sld r0, r3, r31 or r4, r4, r0 srd r3, r3, r6 sld r0, r2, r31 or r3, r3, r0 srad r2, r2, r6	Shift Right Algebraic, N=3 (shift amount < 32) subfic r31, r6, 32 srw r4, r4, r6 slw r0, r3, r31 or r4, r4, r0 srw r3, r3, r6 slw r0, r2, r31 or r3, r3, r0 sraw r2, r2, r6

B.2.1 Notes

The following notes apply to the preceding examples and to the corresponding cases using the other three arithmetic relations (<, ≤, and ≠). They should also be considered when any other use of **fsel** is contemplated.

In these notes, the optimized program is the Book E program shown, and the unoptimized program (not shown) is the corresponding Book E program that uses **fcmpu** and branch conditional instructions instead of **fsel**.

1. The unoptimized program affects FPSCR[VXSNAN] and therefore may cause the system error handler to be invoked if the corresponding exception is enabled; the optimized program does not affect this bit. This property of the optimized program is incompatible with the IEEE standard.
2. The optimized program gives the incorrect result if *a* is a NaN.
3. The optimized program gives the incorrect result if *a* and/or *b* is a NaN (except that it may give the correct result in some cases for the minimum and maximum functions, depending on how those functions are defined to operate on NaNs).

Multiple-Precision Shifts

4. The optimized program gives the incorrect result if a and b are infinities of the same sign. (Here it is assumed that invalid operation exceptions are disabled, in which case the result of the subtraction is a NaN. The analysis is more complicated if invalid operation exceptions are enabled, because in that case the target register of the subtraction is unchanged.)
5. The optimized program affects FPSCR[OX, UX, XX, VXISI], and therefore may cause the system error handler to be invoked if the corresponding exceptions are enabled; the unoptimized program does not affect these bits. This property of the optimized program is incompatible with the IEEE standard.

Appendix C

Guidelines for 32-Bit Book E

This appendix provides guidelines used by 32-bit Book E implementations. Likewise, a set of guidelines is also outlined for software developers. Application software written to these guidelines can be labelled 32-bit Book E applications and can expect to execute properly on all implementations of Book E, both 32-bit and 64-bit implementations.

32-bit Book E implementations execute applications that adhere to the software guidelines for 32-bit Book E software outlined in this appendix and are not expected to properly execute 64-bit Book E applications or any applications not adhering to these guidelines (that is, 64-bit Book E applications).

C.1 64-Bit-Specific Book E Instructions

A subset of Book E instructions are restricted to 64-bit Book E processing. A 32-bit Book E implementation need not implement any of the following instructions. Likewise, neither should 32-bit Book E applications use any of these instructions. All other Book E instructions are either supported directly by the implementation or sufficient infrastructure is provided to enable software emulation of the instructions.

The 64-bit Book E instructions are as follows:

- 64-bit integer arithmetic, compare, shift and rotate instructions
 - **adde64[o]**, **addme64[o]**, **addze64[o]**
 - **subfe64[o]**, **subfme64[o]**, **subfze64[o]**
 - **mulhd**, **mulhdu**, **mulld[o]**, **divd**, **divdu**, **extsw**
 - **cmp** (L=1), **cmpi** (L=1), **cmpl** (L=1), **cmpli** (L=1)
 - **rldcl**, **rldcr**, **rldic**, **rldicl**, **rldicr**, **rldimi**, **sld**, **srad**, **sradi**, **srd**
 - **cntlzd**, **td**, **tdi**
- 64-bit extended addressing branch instructions—**bcctre[l]**, **bce[l][a]**, **bclre[l]**, **be[l][a]**
- 64-bit extended addressing cache management instructions—**dcbae**, **dcbfe**, **dcbie**, **dcbste**, **dcbte**, **dcbstste**, **dcbze**, **icbie**, **icbte**

- 64-bit extended addressing load instructions—**lbze, lbzue, lbzxe, lbzxue, ldarxe, lde, ldue, ldxe, ldxeue, lfde, lfduue, lfdxe, lfdxue, lfse, lfsue, lfsxe, lfsxue, lhae, lhaue, lhaxe, lhaxue, lhbrxe, lhze, lhzue, lhzxe, lhzxue, lwarxe, lwbrxe, lwze, lwzue, lwzxe, lwzxue**
- 64-bit extended addressing store instructions—**stbe, stbue, stbxe, stbxue, stdcxe., stde, stdue, stdxe, stdxue, stfde, stfdue, stfdxe, stfdxue, stfiwxue, stfse, stfsue, stfsxe, stfsxue, sthbrxe, sthe, sthue, sthxe, sthxue, stwbrxe, stwcxe., stwe, stwue, stwxue, stwxue**

C.2 Registers on 32-Bit Book E Implementations

Book E defines 32- and 64-bit registers. All 32-bit registers are supported as defined in Book E. However, except for the 64-bit FPRs, only bits 32–63 of Book E’s 64-bit registers are required to be implemented in hardware in 32-bit Book E implementation. Such 64-bit registers include LR, CTR, 32 GPRs, SRR0, and CSRR0. Book E makes no restrictions regarding implementing a subset of the 64-bit floating-point architecture.

Likewise, other than floating-point instructions, all instructions defined to return a 64-bit result return only bits 32–63 of the result on a 32-bit Book E implementation.

C.3 Addressing on 32-Bit Book E Implementations

Only bits 32–63 of the 64-bit Book E instruction and data memory effective addresses need to be calculated and presented to main memory. Given the only branch and data memory access instructions not included in Section C.1, “64-Bit-Specific Book E Instructions,” are defined to prepend 32 zeros to bits 32–63 of the effective address computation, a 32-bit implementation can bypass the prepending of the 32 zeros when implementing these instructions. For branch to LR and branch to CR instructions, given the LR and CTR are implemented as 32-bit registers, concatenating only 2 zeros to the right of bits 32–61 of these registers is necessary to form the 32-bit branch target address.

The simplest implementation of next sequential instruction address computation suggests allowing effective address computations to wrap from 0xFFFF_FFFC to 0x0000_0000. This wrapping is required of PowerPC implementations. For 32-bit Book E applications, there appears little if any benefit to allowing this wrapping behavior. Book E specifies that the situation where the computation of the next sequential instruction address after address 0xFFFF_FFFC is undefined (note that the next sequential instruction address after address 0xFFFF_FFFC on a 64-bit Book E implementation is 0x0000_0001_0000_0000).

C.4 TLB Fields on 32-bit Book E Implementations

32-bit Book E implementations should support bits 32–53 of the effective page number (EPN) field in the TLB. This size provides support for a 32-bit effective address, which

PowerPC ABIs may have come to expect to be available. 32-bit Book E implementations may support greater than 32-bit real addresses by supporting more than bits 32–53 of the real page number (RPN) field in the TLB.

C.5 32-Bit Book E Software Guidelines

C.5.1 32-Bit Instruction Selection

Any Book E software that uses any of the instructions listed in Section C.1, “64-Bit–Specific Book E Instructions,” is considered 64-bit Book E software, and correct execution cannot be guaranteed on 32-bit Book E implementations. Generally speaking, 32-bit software should avoid instructions that depend on any particular setting of bits 0–31 of any 64-bit application-accessible system register, including GPRs, for producing the correct 32-bit results. Context switching is not required to preserve the upper 32 bits of application-accessible 64-bit system registers and insertion of arbitrary settings of those upper 32 bits at arbitrary times during the execution of the 32-bit application must not affect the final result.

C.5.2 32-Bit Addressing

Book E provides a complete set of data memory access instructions that perform a modulo 2^{32} on the computed effective address and then prepend 32 zeros to produce the full 64-bit address. Book E also provides a complete set of branch instructions that perform a modulo 2^{32} on the computed branch target effective address and then prepend 32 zeros to produce the full 64-bit branch target address. On a 32-bit Book E implementation, these instructions are executed as defined, but without prepending the 32 zeros (only the low-order 32 bits of the address are calculated). On a 64-bit implementation, executing these instructions as defined provides the effect of restricting the application to lowest 32-bit address space.

However, there is one exception. Next sequential instruction address computations (not a taken branch) are not defined for 32-bit Book E applications when the current instruction address is 0xFFFF_FFFC. On a 32-bit Book E implementation, the instruction address could simply wrap to 0x0000_0000, providing the same effect that is required in the PowerPC Architecture. However, when the 32-bit Book E application is executed on a 64-bit Book E implementation, the next sequential instruction address calculated will be 0x0000_0001_0000_0000 and not 0x0000_0000_0000_0000. To avoid this problem the 32-bit Book E application must either avoid this situation by not allowing code to span this address boundary, or requiring a branch absolute to address 0 be placed at address 0xFFFF_FFFC to emulate the wrap. Either of these approaches allows the application to execute on 32-bit and 64-bit Book E implementations.

Index

Numerics

32-bit Book E software guidelines, C-3
64-bit-specific Book E instructions, C-1

A

Access permission
 read, 11-29

Accumulator, 2-59, 10-6

addi, A-22

addis, A-22

Address translation

 Book E, 11-23
 effective to real translation, 11-23
 effective-to-real, 13-3
 overview, 1-26

Address translation and the Motorola Book E
 standard, 11-32

Addressing conventions

 mapping examples, 11-7

Addressing modes

 32-bit Book E implementations, C-2
 branch conditional
 absolute, 3-35
 count register, 3-36
 link register, 3-35
 relative, 3-33
 branch relative, 3-32
 branch to absolute, 3-34
 data memory, 3-14
 instruction memory, 3-14
 register indirect
 integer, 3-28
 with immediate index, integer, 3-26
 with index, integer, 3-27

Alignment

 DEAR-caused, 5-4
 effective address, 5-8
 exception priorities, 5-38, 5-39
 exceptions, 5-3, 5-21
 interrupt, 5-10, 5-12, 5-21, 5-22, 5-35
 interrupt order, 5-36
 interrupt type, 5-6
 misaligned accesses, 3-1

 unaligned load/store, 5-33

Altivec technology LRU instructions, 12-26

APUs

 accumulator, 2-59, 10-6
 allocated bits in Book E-defined registers, 10-5
 branch target buffer (BPU), 10-13
 BTB instructions, 10-14
 BTB programming model, 10-14
 BTB registers, 10-14
 BUCSR, 10-14
 cache block lock and unlock, 3-71
 programming model, 10-15
 requirements, 10-14
 cache locking interrupt and register model, 10-16
 context switching, 10-7
 exception priorities, 10-11
 floating-point data exception, 10-10
 floating-point instruction implementation, 3-2
 IEEE 754 compliance, 10-9
 instruction select, 10-1
 interrupt register, 5-24
 IVORs, 10-8
 machine check
 instruction model, 10-17
 interrupt, 10-17
 programming and register model, 10-16
 performance monitor, 3-71, 7-2, 7-8
 performance monitor registers, 7-2
 PowerPC UISA and Book E instructions, 10-8
 programming model, 10-2
 signal processing, 10-4
 SPEFSCR, 10-5
 SPFPU operations, 10-9
 SPU
 exceptions, 10-9
 functions, 10-11
 instruction model, 10-8
 programming model, 10-4
 register model, 10-4
 vector
 floating-point round exception, 10-10
 unavailable exception, 10-10
 using **isel**, 10-2
 vector alignment exception, 10-11

- Arithmetic instructions
 - integer, 3-6, 3-7, 3-22
- Assumed TLB entries, 13-22
- Atomic memory references, 11-17
- Atomic update memory, 3-30
- Atomic update memory references, 1-28
- Atomic update primitives, 3-46

B

- Big-endian pages, 11-7
- BO encodings, 3-37
- Book E
 - 32-bit
 - addressing, C-3
 - implementations, 2-3
 - instruction selection, C-3
 - software guidelines, C-3
 - address translation, 11-23
 - cache control instructions, 11-10
 - debug mode deviations, 8-1
 - implementation-specific instructions, 3-57
 - instruction listing, 3-73
 - instructions and UISA, 10-8
 - interrupt registers, 2-23
 - SPR model, 2-66
 - TLB concept, 11-21
 - user-level instructions, 3-22
- Book E architecture
 - address spaces *see also* Instruction data address spaces
 - MMU, 11-17
 - virtual address, 11-19
- Book E MMU
 - permission attribute comparison, 11-26
- BookE MMU
 - process ID, 11-21
- Boundedly undefined, definition, 3-7
- Branch instructions
 - address calculation, 3-32
 - basic mnemonics, A-6
 - BO operand encodings, 3-37
 - completion, 4-20
 - condition register logical, 3-40, A-14
 - conditional
 - absolute addressing mode, 3-35
 - control, 3-37
 - relative addressing mode, 3-33
 - CTR addressing mode, 3-36
 - folding, 4-44
 - list, 3-39, 4-44
 - LR addressing mode, 3-35
 - relative addressing mode, 3-32
 - resolution definition, 4-1

- simplified mnemonics, A-6
- system linkage, 3-40, 3-54
- trap, 3-40
- Branch prediction
 - definition, 4-1
 - instruction model, 3-58
- Branch processing unit (BPU)
 - execution timing, 4-20
- Branch taken (BRT) debug event, 8-11
- Branch target buffer (BTB)
 - entry address register (BBEAR), 2-32
 - instructions, 3-70, 10-14
 - registers, 2-32, 10-14
 - target address register (BBTAR), 2-33
- Branch unit control and status register 0 (BUCSR0), 2-32, 2-33, 10-14
- Byte ordering, 3-15

C

- Cache
 - allocation on misses, 12-23
 - AltiVec technology LRU instruction support, 12-26
 - arbitration, 4-14
 - block lock and unlock APU, 3-71, 10-15
 - cacheable loads and LSU, 12-4
 - caching-inhibited accesses (I bit), 11-2
 - caching-inhibited memory, 11-3
 - coherency, 11-1
 - coherant memory, 4-15
 - memory bit (M bit), 11-2
 - control
 - dcbi**, 3-55
 - dcbt**, 3-52
 - instructions, 11-10, 12-15, 12-16
 - overview, 12-15
 - control instructions, overview, 1-28, 12-15
 - data block push operation, 12-24
 - data cache
 - block fill operations, 12-22
 - block push operations, 12-24
 - configuration, 12-5
 - enabling/disabling with HID0, 12-18
 - fills, 12-22
 - data enabling/disabling with HID0, 12-19
 - data locking using HID0, 12-21
 - data organization diagram, 12-6
 - data status bits, 12-8, 12-9, 12-16
 - data store hit, 12-24
 - endianness, 11-6
 - enforcing store ordering, 11-12, 12-15
 - guarded accesses
 - G bit, 11-2
 - guarded memory accesses, 12-13

- guarded memory bit (G bit), 11-2
- instruction cache
 - block fill operations, 12-23
 - coherency model, 12-11
 - fills
 - organization, 12-7
 - organization diagram, 12-7
- L1 cache
 - block replacement, 12-24
 - configuration register 0, 2-39
 - configuration register 1, 2-40
 - data cache flushing, 12-21
 - data organization, 12-5
 - features list, 12-1
 - instruction organization, 12-7
 - operation, 12-22
 - organizations, 12-5
- L1 data cache
 - configuration, 12-5
- L2 cache
 - data fills, 12-22
- load/store operations, processor initiated, 12-14
- load/store unit (LSU)
 - general, 4-29, 12-3
 - store queues, 4-28, 12-4
- locking and PLRU, 12-26
- locking APU register and interrupt model, 10-16
- LSU store queues, 4-28, 12-4
- miss and reload operations, 12-22
- on-chip implementation, 1-19
- operations
 - data block fill, 12-22
 - data block push, 12-24
 - instruction fill
 - load/store unit (LSU), 12-14
- overview, 12-1
- PLRU replacement, 12-25
- store miss merging, 12-23
- write-through attribute, 11-3
- write-through mode (W bit), 11-2
- Cache/memory attributes
 - access and sequential consistency, 12-14
 - coherency paradoxes and WIMGE, 11-9
 - G bit of WIMGE, 13-14
 - inhibited loads and stores, 12-13
 - memory coherence required, 11-3
 - misaligned accesses and the endian (E) bit, 12-13
 - speculative accesses, 12-13
 - subsystem block diagram, 4-27, 12-3
 - WIMGE general, 12-12
 - write-back stores, 12-12
 - write-through stores, 12-13
- Classes of instructions, 3-7
- Code, self-modifying, 3-26, 11-10
- Coherency
 - caching and memory, 4-15
 - memory and cache, 11-1
 - paradoxes, 11-9
- Compare and swap, B-4
- Compare instructions
 - simplified mnemonics, A-3
- Completion
 - considerations, 4-15
 - definition, 4-2
 - groupings, 4-46
 - instruction, 4-10
 - resource requirements, 4-45
- Conditional branch control, 3-37
- Context switching APU, 10-7
- Context synchronization, 3-58
- Conventions, xxv
- Conventions in timing terminology, 4-1
- Core complex bus (CCB)
 - overview, 1-29
- Core complex MMU registers, 13-23
- CR (condition register)
 - bit and identification symbols, A-1
 - execution latencies, 4-32, 4-37
 - logical instructions, 3-40, A-14
 - simplified mnemonics, A-14
- Critical input interrupt, 5-14
- Critical save/restore register 0, 2-24, 5-4
- Critical save/store register 1, 2-24
- Crossing page boundaries, 11-31
- CSRR0 (critical save/restore register 0), 2-24, 5-4
- CSRR1 (critical save/restore register 1), 2-24, 5-4
- CTR (count register), branch conditional, 3-36

D

- Data
 - exception address register, 5-4
 - exception address register (DEAR), 2-25
 - memory addressing modes, 3-14
 - organization in memory, 3-1
 - TLB error interrupt, 5-27
 - transfers, 3-1
- Data address compare
 - debug event, 8-8
 - mode, 8-9
 - read/write enable, 8-8
 - user/supervisor mode, 8-9
- Data address spaces
 - Book E MMU, 11-20
- Data cache
 - block fill operations, 12-22
 - block push operation, 12-24
 - block push operations, 12-24

- configuration, 12-5
- enabling/disabling, 12-18
- Data storage interrupt (DSI), 5-17
- dcbi**, 3-55
- dcbt**, 3-52
- DEAR, 2-25, 5-4
- Debug interrupt, 5-29
- Debug support
 - hardware facilities, 8-2
 - overview, 8-1
 - software facilities, 8-2
 - TAP controller, 8-2
- Decrementer auto-reload register (DECAR), 2-22, 9-10
- Decrementer interrupt, 5-24
- Decrementer register (DEC), 9-10
- Disabling timer, 6-3
- Dispatch
 - considerations, 4-15
 - definition, 4-2
 - notation, 4-10
 - unit resource requirements, 4-44

E

- Effective address (EA)
 - calculation, 3-13
 - debug mode, 8-6, 8-9
 - general, 4-30
 - loads/stores, 3-26
- Effective-to-real address translation, 13-3
- Embedded floating-point instructions, 1-11, 3-2, 3-68
- Endianness
 - byte ordering, 3-15
 - differences, 1-32
 - memory coherency, 11-6
- ESR, 2-26, 5-4
- Event counting, 7-8
- Event selection, 7-10
- Examples
 - programming, 11-15
- Exception handling
 - asynchronous interrupts, 5-9
 - extended model, 1-19
 - overview, 1-31
- Exception priorities
 - Book E, 5-36
 - integer loads and stores, 5-38
 - list, 10-11
 - privileged instructions, 5-39
 - system call instruction, 5-40
 - trap instructions, 5-40
- Exception syndrome register, 5-4
- Exception vector floating-point round, 10-10

- Exceptions
 - alignment, 5-21
 - critical input, 5-14
 - DSI, 11-2, 11-31, 11-47, 13-17, 13-29
 - illegal instruction, 5-23
 - instruction storage, 5-19
 - instruction TLB error, 5-29
 - instruction-related exceptions, 3-20
 - machine check, 5-15
 - Motorola Book E TLB miss, 11-45
 - overview, 5-1
 - priorities, 5-36, 5-39, 10-11
 - program, 5-22
 - SPU vector floating-point data, 10-10
 - syndrome register, 2-26
 - system call, 5-24
 - TLB miss, 11-18, 11-45, 13-2, 13-10, 13-20
 - trap exception, 5-23
 - vector alignment, 10-11
 - vector floating-point data, 5-31
- Execute access permission, 11-28
- Execution
 - instructions, 4-10
 - synchronization, 3-20
 - timing, FPU, 4-29
- Execution model
 - out-of-order execution, 11-5
- Execution units
 - timing examples, 4-20
 - timing, LSU, 4-30

F

- Features list, 1-5
- Fetch
 - and add, B-3
 - and AND, B-3
 - and branch considerations, 4-44
 - and no-op, B-2
 - and store, B-3
 - definition, 4-2
 - instruction timing, 4-10
- Finish definition, 4-2
- Fixed-interval timer, 9-1, 9-12
- Fixed-interval timer interrupt, 5-25, 9-14
- Floating-point
 - execution timing, 4-29

G

- General-purpose registers (GPRs), 2-9
- Global control register 0, 1-29
- Global control register 0 (PMGC0), 2-61
- GPR issue queue (GIQ), 4-45
- Guarded accesses

- G bit, 13-14
- G bit of WIMGE, 13-14
- Guarded attribute (G)
 - guarded memory, 11-5
 - out-of-order execution, 11-5
- Guarded memory
 - speculative accesses, 11-6

H

- Hardware implementation-dependent register 0 (HID0), 2-35, 9-2
 - data cache enabling/disabling, 12-18, 12-19, 12-21
 - locking, 12-21
- Hardware implementation-dependent register 1 (HID1), 2-36, 12-8

I

- IEEE 754 compliance, 10-9
- Import barriers, 11-16
- Imprecise interrupts, 5-10
- Instruction address compare
 - debug event, 2-50
 - mode, 8-7
 - user/supervisor mode, 8-6
- Instruction address resume register (IARR), 2-65
- Instruction address spaces
 - Book E MMU, 11-20
- Instruction cache, *see* Cache, instruction
- Instruction fetch
 - stages, 1-15, 4-7
 - timing, 4-13
- Instruction flow, 1-13, 4-12
- Instruction memory addressing modes, 3-14
- Instruction model APU machine check, 10-17
- Instruction pipeline stages
 - complete, 1-17, 4-9
 - decode/dispatch, 1-16, 4-7
 - execute, 1-17, 4-8
 - general, 4-6
 - instruction fetch, 1-15, 4-7
 - issue queues (FIQ, VIQ, GIQ), 1-16, 4-7
 - write-back, 1-17, 4-9
- Instruction select (**isel**) APU, 3-70, 10-1
- Instruction set
 - compatibility, 1-31
 - overview, 1-11
- Instruction storage interrupt (ISI)
 - conditions, 5-19
- Instruction timing
 - BPU execution timing, 4-20
 - completion, 4-10
 - CR execution latencies, 4-32, 4-37
 - dispatch, 4-10, 4-15

- execute, 4-10
- execution units, 4-20
- fetch, 4-10
- flow, 4-12
- issue, 4-10
- IU execution timing, 4-29
- latency summary, 4-31
- LSU execution latencies, 4-34
- memory coherency and the cache, 4-15
- memory performance considerations, 4-31
- overview, 4-4
- pipelining
 - load/store, 4-47
 - misalignment effects, 4-47
- rename register operation, 4-16
- scheduling guidelines, 4-43
- stage definition, 4-3
- terminology, 4-1
- write-back, 4-10
- Instruction TLB error interrupt settings, 11-45
- Instructions
 - APU performance monitor, 7-8
 - APU programming model, 10-2
- Book E
 - 64-bit-specific, C-1
 - implementation-specific, 3-57
 - listing, 3-73
 - user-level, 3-22
- boundedly undefined, 3-7
- branch, 4-20
 - address calculation, 3-32
 - conditional
 - absolute addressing mode, 3-35
 - control, 3-37
 - CTR addressing mode, 3-36
 - LR addressing mode, 3-35
 - relative addressing mode, 3-33
 - fetch/branch considerations, 4-44
 - predicting and resolution, 4-21
 - simplified mnemonics, A-5
- branch target buffer (BTB), 3-70, 10-14
- cache control
 - e500 core, 1-28
 - instructions, 11-10, 12-16
- classes
 - one of four, 3-7
 - reserved-illegal, 3-11
 - reserved-nop, 3-11
- condition register logical, 3-40
- conditional control, 3-37
- exceptions, 3-20, 5-36
- execution latencies, *see* Instruction timing
- execution synchronization, 3-20
- fetching, 3-58

- floating-point, 3-2, 3-68
 - compare, A-3
- flow diagram, 4-6
- integer
 - arithmetic, 3-22
 - compare, 3-23, A-3
 - load/store multiple, 3-31
 - logical, 3-24
 - rotate and shift, 3-24
 - rotate/shift, A-3
 - store, 3-30
- load and store
 - address generation
 - integer, 3-26
 - byte reverse, 3-31
 - execution latencies, 4-34
 - integer
 - multiple, 3-31
 - store, 3-30
 - memory synchronization, 3-45
 - misalignment handling, 3-26
- LRU, 12-26
- memory control, 3-51, 3-54
- memory synchronization, 3-60
- new e500, 3-57
- no-op, A-21
- partially executed, 5-32
- PowerPC
 - UISA and Book E, 10-8
- processor control, 3-41
- refetch serialization, 4-18
- scalar floating-point, 1-11
- select APU, 3-70, 10-1
- serialization, 4-18, 4-46
- set overview, 3-21
- set summary, 3-6
- SPU (signal processing APU)
 - floating-point, 3-68
 - general, 3-61
 - vector, 3-61
- store serialization, 4-18
- system
 - call, 5-40
 - linkage, 3-40
 - register instruction latencies, 4-31
- TLBs (translation lookaside buffer)
 - invalidate, 13-17
 - read entry, 13-15
 - search, 13-17
 - synchronize, 13-19
- TLBs (translation lookaside buffers)
 - write entry, 13-16
- trap, 3-40
- unsupported, 3-2
 - update, 3-59
- Integer
 - arithmetic instructions, 3-22
 - compare instructions, 3-23, A-3
 - exception registers, 2-9
 - loads and stores, 3-7, 3-26, 3-29, 5-38
 - logical instructions, 3-24
 - rotate/shift instructions, 3-24, A-3
 - store instructions, 3-30
 - unit execution timing, 4-29
- Interrupt classes
 - asynchronous, 5-9
 - categories, 1-20
 - synchronous, 5-10
- Interrupt processing, 5-11
- Interrupt registers
 - alignment settings, 5-22
 - critical input settings, 5-15
 - data storage settings, 5-18
 - data TLB error settings, 5-27
 - debug settings, 5-30
 - decrementer settings, 5-25
 - defined by Book E, 2-23, 5-4
 - e500-specific, 2-28
 - external input settings, 5-20
 - fixed-interval timer settings, 5-25
 - general, 5-3
 - instruction storage settings, 5-19
 - instruction TLB settings, 5-29
 - overview, 1-22
 - program settings, 5-23
 - system call settings, 5-24
 - vector prefix (IVPR), 5-4
 - watchdog timer settings, 5-26
- Interrupts
 - alignment, 5-21
 - and exception handling registers
 - DEAR, 5-4
 - ESR, 2-26, 5-4
 - MSR, 2-14
 - critical input, 5-14
 - critical, non-critical classes, 5-8
 - data storage, 5-17
 - data TLB error, 5-27
 - debug, 5-29
 - decrementer, 5-24
 - fixed-interval timer, 5-25
 - imprecise, 5-10
 - instruction partially executed, 5-32
 - instruction storage, 5-19
 - instruction TLB error, 5-29
 - IVORs, 5-4
 - machine check, 5-15, 10-17
 - masking, 5-34

- order, 5-35
- ordering, 5-34
- overview, 5-1
- precise, 5-10
- program, 5-22
- program exception, 5-23
- program illegal instruction exception, 5-23
- system call, 5-24
- vector offset registers, 2-25, 5-5
- vector prefix register, 2-25
- watchdog timer, 5-26

IPROT invalidation protection, 13-10

isel, improved conditionals using, 10-2

ISI, 5-19

Issue

- definition, 4-2
- illustration, 4-10

isync, 3-45

IU1 considerations, 4-46

IU2 considerations, 4-47

IVORs, 2-25, 5-5

IVPR, 2-25, 5-4

J

JTAG test

- clock (TCK) input, 8-3
- data
 - input (TDI), 8-3
 - output (TDO), 8-4
- mode select (TMS) input, 8-4
- reset (TRST), input, 8-4

L

L1 and L2 TLB access times, 4-13, 13-13

L1 cache

- block replacement selection, 12-24
- configuration register 0 (L1CFG0), 2-39
- configuration register 1 (L1CFG1), 2-40
- data flushing, 12-21
- data organization, 12-5
- operation, 12-22
- organizations, 12-5

L1 TLB arrays, 13-8

L2 TLB arrays, 13-9

Latency definition, 4-2

Link stack registers, 4-13

List insertion, B-5

Little-endian pages, 11-7

Load/store

- misalignment handling, 3-26
- multiple instructions, 3-31

Load/store unit (LSU)

- address generation, 3-26

- byte reverse instructions, 3-31
- cacheable loads, 12-4
- execution latencies, 4-34
- execution timing, 4-30
- general, 4-27, 4-29, 12-3
- integer
 - store instructions, 3-30
- operations, 12-14
- store queues, 4-28, 12-4

Local control A registers (PMLCa0–PMLCa3), 2-62

Local control B registers (PMLCb0–PMLCb3), 2-63

Local control registers

- A 0 – 3 (PMLCa0–PMLCa3), 2-62
- B 0 – 3 (PMLCb0–PMLCb3), 2-63

overview, 1-30

Lock acquisition, 11-16

Lock acquisition and release, B-4

LR, 2-13

M

Machine check interrupt, 1-21, 5-2, 5-15, 10-17

Machine state register (MSR), 2-14

MAS register updates for exceptions, tlbsx, and tlbre, 11-47

MAS registers, 13-24

- updates, 13-29

Memory

- shared, 11-11
- synchronization, 3-45
- synchronization instructions, 3-60

Memory access

- alignment support, 3-58
- ordering, 11-11
- ordering overview, 1-28
- sequential consistency, 12-14

Memory access ordering, 11-11

- programming considerations, 11-13

Memory accesses

- speculative and out-of-order, 11-4

Memory cache attributes, 11-8

Memory coherency

- e500 core, 1-27
- general, 11-1
- required, 11-3

Memory control instructions

- description, 3-51, 3-54

Memory management

- overview, 1-24, 1-32
- unit implementation, 13-1

Memory protection

- string instructions, 11-31

Memory references, atomic update, 1-28

Memory subsystem

- data treatment and WIMGE, 11-22
- load condition, 11-14
- overview, 1-31
- Memory/cache access attributes (WIMGE bits), 11-2
- Memory/cache attributes
 - guarded, 11-4
- Memory/cache coherency
 - programming examples, 11-15
- Misalignment in accesses, 3-1
- Mismatched memory cache attributes, 11-8
- MMU
 - assist register field updates, 5-28, 13-29
 - assist registers (MAS0–MAS4 and MAS6), 2-45
 - assist registers (MAS0–MAS6), 11-35
 - assist registers (MAS1–MAS4 and MAS6), 1-26
 - Book E address spaces, 11-19
 - Book E architecture, 11-17
 - Book E data address spaces, 11-20
 - Book E instruction address spaces, 11-20
 - Book E process ID, 11-21
 - Book E virtual address, 11-19
 - MAS register updates, 13-29
 - Motorola Book E TLB
 - configuration information, 11-48
 - configuration registers, 11-49
 - miss exception, 11-45
 - reading entries, 11-36
 - registers, 2-41
 - TLB
 - arrays summary, 13-6
 - coherency, 1-27
 - consistency between L1, L2, 13-12
 - entry field definitions, 13-14
 - entry reload features, 13-2, 13-20
 - fields on 32-bit Book E implementations, C-2
 - instructions implementation, 13-15
 - invalidate all address encoding, 13-19
 - invalidate broadcast enabling, 13-19
 - invalidate instruction, 13-17
 - L1 arrays, 13-8
 - L2 arrays, 13-9, 13-13
 - management instructions, 12-10
 - miss exception updates, 13-21
 - read entry instruction, 13-15
 - search instruction, 13-17
 - selection for **tlbivax**, 13-19
 - states after reset, 13-22
 - synchronize instruction, 13-19
 - translation reload interrupt routines, 13-21
 - write entry instruction, 13-16
- Mnemonics
 - recommended, A-21
 - simplified, A-1
- Motorola Book E
 - address translation, 11-32
 - algorithms, 11-34
 - data
 - storage interrupt (DSI), 11-47
 - TLB error interrupt, 11-45
 - exceptions
 - TLB miss, 11-45
 - instruction storage interrupt (ISI)
 - settings for permissions violation, 11-47
 - IPROT TLB entry field, 11-35
 - MAS registers
 - exception handling, 11-44
 - formats and bit definitions, 11-39
 - match criteria for TLB entries, 11-33
 - MMU
 - assist (MAS) registers, 11-38
 - assist MAS register field updates, 11-47
 - configuration register (MMUCFG), 11-49
 - TLB
 - entry invalidations, 11-36
 - reading entries, 11-36
 - variable PID registers and sharing, 11-32
 - MMU architecture, 11-31
 - page sharing with PID registers, 11-32
 - permissions violation
 - exception types, 11-46
 - PID registers, 11-32
 - process ID registers (PID)
 - contents set, 11-46
 - TLB entries
 - general, 11-35
 - match criteria, 11-33
 - reading, 11-35
 - writing, 11-36
 - TLB miss exception types, 11-45
 - TLBs (translation lookaside buffers)
 - configuration information, 11-48
 - configuration registers (TLB1FGn), 11-49
 - entry invalidations, 11-36
 - reading entries, 11-36
- Multiple-precision shifts, 3-25, B-7

N

No-op, A-21

O

- On-chip cache implementation, 1-19, 12-1
- Operands
 - BO encodings, 3-37
 - conventions, 3-1
 - placement and performance, 4-30
- Operations
 - data cache block push, 12-24

- instruction cache block fill
- Out-of-order execution, 11-5
- Overflow (OV), 2-10

P

- Page characteristics
 - e500 core, 1-28
 - size, 11-22
- Page size
 - effective address bits compared, 11-25
 - real address generation, 11-27
- Pages
 - big-endian, 11-7
 - little-endian, 11-7
- Performance monitor
 - event counting, 7-8
 - event selecting, 7-10
 - purposes, 1-5, 7-1, 10-2
- Performance monitor APU, 3-71
 - counter registers, 1-30, 2-64
 - general, 7-1, 10-2
 - instructions, 7-8
 - programming model, 7-2
 - registers, 7-2
- Performance monitoring, overview, 1-29
- Performed loads and stores, 12-14
- Permission attributes, 11-28
- Permission control and cache management
 - instructions, 11-30
- Permissions
 - execute access, 11-28
 - maintaining history, 11-31
 - write access, 11-29
- PID, 13-22
- Pipeline
 - instruction timing definition, 4-2
 - stages and events, 4-10
 - superscalar diagram, 4-5
- PIR, 2-16, 2-41
- PLRU, L1 caches
 - bit updates, 12-25
 - replacement, 12-24
- Position-independent code example, 4-44
- Power management
 - control bits, 6-5
 - down state snooping, 6-4
 - full on, 6-2
 - further reduction, 6-3
 - halted, 6-2
 - interrupts, 6-3
 - power down, 6-2
 - signals, 6-4
 - software considerations, 6-5

- states, 6-1
- PowerPC Architecture
 - legacy support, 1-30
 - memory accesses and sequential consistency, 12-14
 - UISA instructions, 10-8
- PowerPC architecture
 - user instruction set architecture (UISA), xxi
- PowerPC Architecture legacy support
 - overview, 1-30
- Precise interrupts, 5-10
- Preserved special-purpose registers, 2-67
- Privileged instructions, 5-39
- Process ID, 11-21
 - Motorola Book E, 11-32
- Process ID registers (PID)
 - core complex, 13-4
 - modifying, 13-22
 - Motorola Book E sharing, 11-32
 - overview, 1-27
 - TLB entries, 13-18
- Processor control instructions, 3-41
- Processor ID register (PIR), 2-41
- Program order definition, 4-3
- Programmable page characteristics, 1-28
- Programming model
 - APU performance monitor, 7-2, 10-4
 - e500 core, 1-18, 2-2
 - overview, 1-17
 - updating, 4-22
- PVR, 2-17

R

- Read access permission, 11-29
- Register model
 - 32-bit Book E implementations, 2-3
 - allocated SPRs, 2-68
 - branch operations, 2-10
 - condition register (CR), 2-10
 - count register (CTR), 2-14
 - CR setting
 - compare instructions, 2-13
 - integer instructions, 2-12
 - store conditional instructions, 2-12
 - debug support, 8-2
 - invalid SPR references, 2-66
 - reserved SPRs, 2-68
 - signal processing APU (SPU), 10-4
 - synchronization requirements for SPRs, 2-66
 - time base, 2-21
- Registers
 - allocated bits in Book E signal processing APU (SPU), 10-5
 - APU performance monitor, 7-2

- branch buffer entry address, 2-32
- branch buffer target address (BBTAR), 2-33
- branch operations, 2-10
- branch target buffer (BTB), 2-32, 10-14
- branch unit control and status 0 (BUCSR0), 2-32, 2-33, 10-14
- condition, 2-10
- core complex MMU, 13-23
- count (CTR), 2-14
- critical save/restore 0 (CSRR0), 2-24
- critical save/restore1 (CSRR1), 2-24, 5-4
- CSRR0, 5-4
- data exception address (DEAR), 2-25
- DEAR, 5-4
- decrementer (DEC), 2-22, 9-10
- decrementer auto-reload (DECAR), 2-22, 9-10
- e500 core assist, 1-26
- ESR, 5-4
- exception syndrome (ESR), 2-26
- general-purpose, 2-9
- global control, 1-29
- hardware implementation-dependent
 - 0 (HID0), 2-35, 9-2
 - 1 (HID1), 2-36
- implementation-specific
 - MMCR1, 7-4, 7-5, 7-6
- instruction address resume (IARR), 2-65
- integer exception (XER), 2-9
- interrupt, 5-3
 - defined by Book E, 2-23
 - general, 2-23
 - vector offset (IVORs), 2-25, 5-5, 10-8
 - vector prefix (IVPR), 2-25, 5-4
- interrupt vector prefix, 5-4
- L1 cache configuration 0 (L1CFG0), 2-39, 2-40
- link (LR), 2-13, 4-13
- local control, 1-30
- local control A (PMLCa0–PMLCa3), 2-62
- local control B (PMLCb0–PMLCb3), 2-63
- machine check save/store 1 (MCSRR1), 2-28
- machine state (MSR), 2-14
- MAS
 - e500, 13-24
 - updates, 13-29
- MMU
 - Motorola Book E configuration, 11-49
 - overview, 2-41
- Motorola Book E
 - MAS
 - exception handling, 11-44
 - formats, 11-39
 - MMU
 - assist, 11-38
 - TLB configuration, 11-49

- Motorola Book E PID, 11-32
- performance monitor counter, 1-30, 2-64
- PID, 13-22
- preserved special-purpose, 2-67
- process ID, 1-27
- processor
 - control, 2-14
 - ID (PIR), 2-16, 2-41
 - version (PVR), 2-17
- rename operation, 4-16
- save/restore 0 (SRR0), 2-23
- save/restore 1 (SRR1), 2-23
- signal processing and embedded floating-point
 - status and control, 10-5
- softstop, 2-65
- software-use SPRs, USPRG0, 2-31
- special-purpose (SPRs)
 - encodings, 3-71
 - general, 2-4
- SPRG0-SPRG7, 2-31
- SRR0, 5-4
- SRR1, 5-4
- supervisor-level
 - MMCR1, 7-4, 7-5, 7-6
- timer
 - control (TCR), 2-19, 9-2
 - general, 2-17, 9-2
 - status (TSR), 2-20, 9-4
- user global control 0 (UPMG0), 2-62
- user local control A (UPMLCa0–UPMLCa3), 2-63
- user local control B (UPMLCb0–UPMLCb3), 2-64
- user performance monitor counter
 - (UPMC0–UPMC3), 2-65
- Rename buffer, definition, 4-3
- Rename register operation, 4-16
- Reservation station, definition, 4-3
- Reserved-illegal instruction, 3-11
- Reserved-nop instruction, 3-11
- Reset
 - common vector, 1-32
 - system generation, 5-10
- Retirement, definition, 4-3
- Return debug event, 8-12
- Rotate/shift instructions, 3-24, A-3
- RTLBO array, reading entries, 13-16
- RTLBI array, reading entries, 13-16

S

- Save/restore register 0 (SRR0), 5-4
- Save/restore register 1 (SRR1), 2-23, 5-4
- Searching the TLBI and TLBO arrays, 13-17
- Self-modifying code, 11-10
- Sequential consistency of memory accesses, 12-14

- Serialization
 - instructions, 4-18, 4-46
- Shared memory, 11-11
 - acquire lock and import, 11-16
 - obtain pointer and import, 11-16
- Shared memory multiprocessor support, 12-10
- SHAREN
 - page attribute, 13-11
 - TLB entry bit, 13-15
- Sharing TLB entries, 11-33
- Shift/rotate instructions, 3-24, A-3
- Signal processing and embedded floating-point status
 - and control register (SPEFSCR), 10-5
- Signal processing APU (SPU)
 - allocated bits in Book E–defined registers, 10-5
 - exceptions, 10-9
 - functions, 10-11
 - general, 10-4
 - instruction model, 10-8
 - instructions, 3-61
 - interrupt vector offset registers, 10-8
 - programming model, 10-4
 - register model, 10-4
 - signal-processing floating-point operations, 10-9
 - unavailable exception, 10-10
 - vector
 - floating-point data exception, 10-10
- Signal-processing floating-point SPU (SPFPU)
 - operations, 10-9
- Signals
 - non-protocol specific
 - TCK (JTAG test clock), 8-3
 - TCK (JTAG test clock), 8-3
- Simplified mnemonics, 3-56
 - branch instructions, A-5
 - compare instructions, A-3
 - CR logical instructions, A-14
 - recommended, A-21
 - rotate and shift, A-3
 - special-purpose registers (SPRs), A-17
 - subtract instructions, A-2
 - trap instructions, A-15
- Snoop signaling, 12-11
- Softstop registers, 2-65
- Special-purpose registers (SPRs), A-17
- SPR model, Book E, 2-66
- SPRG0-SPRG7, 2-31
- SPRs, 2-4
- SPU (signal procesing APU)
 - floating-point instructions, 3-68
 - vector instructions, 3-61
- SRR0, 2-23, 5-4
- SRR1, 2-23, 5-4
- SRU (system register unit) execution latencies, 4-31
- Stage, definition, 4-3
- Stall, definition, 4-3
- Store hit to a data cache block marked shared, 12-24
- Store miss merging, 12-23
- Structure mapping examples, 11-7
- Subtract instructions, A-2
- Summary overflow (SO), 2-10
- Superscalar, definition, 4-4
- Synchronization
 - context, 3-58
 - general, B-1
 - memory instructions, 3-45
 - primitives, B-2
- Synchronous interrupts, 5-10
- System
 - call instruction, 5-40
 - call interrupt, 5-24
 - linkage instructions, 3-40, 3-54

T

- TAP
 - enable (TAP_EN), input, 8-4
 - interface signals, 8-2
- TCK (JTAG test clock) signal, 8-3
- TCR, 2-19
- Terminology conventions, xxvi
- Test and set function, B-3
- Throughput, definition, 4-4
- Time base
 - computing time of day, 9-7
 - reading, 9-7
 - writing, 9-7
- Timers
 - control register (TCR), 9-2
 - e500 implementation, 9-9
 - facilities, 9-1
 - fixed-interval, 9-12
 - freezing facilities, 9-15
 - non-constant update frequency, 9-9
 - registers, 9-2
 - status register (TSR), 9-4
 - time base, 9-4
 - watchdog, 9-13
- TLB, 13-10
- TLB entries
 - PID, 13-18
 - sharing, 11-33
- TLBs (translation lookaside buffers)
 - coherency, 1-27
 - consistency between L1 and L2, 13-12
 - entry field definitions, 13-14
 - entry reload features, 13-2, 13-20
 - fields on 32-bit Book E implementations, C-2

- general, 13-6
- instructions implementation, 13-15
- invalidate (**tlbivax**) instruction, 13-17
- invalidate all address encoding for **tlbivax**, 13-19
- invalidate broadcast enabling, 13-19
- L1 arrays, 13-8
- L2 arrays
 - TLB0, 13-9
 - TLB1, 13-13
- management instructions, 12-10
- miss exception MAS register settings, 11-45
- miss exception not taken, 13-10
- miss exception updates, 13-21
- read entry (**tlbre**) instruction, 13-15
- reload interrupt routines, 13-21
- search (**tlbsx**) instruction, 13-17
- selection for **tlbivax**, 13-19
- states after reset, 13-22
- synchronize **tlbsync** instruction, 13-19
- write entry (**tlbwe**) instruction, 13-16
- TLM selected (TLMSEL), output, 8-4
- TO operand, A-17
- Trap instructions, 3-40, 5-40, A-15
- TSR, 2-20

U

- UMMCR1 (user monitor mode control register 1), 7-6
- Unconditional debug event, 8-5, 8-13
- Unsupported instructions and instruction forms, 3-2
- Update instructions, 3-59
- User global control register 0 (UPMGC0), 2-62
- User local control A registers
 - (UPMLCa0–UPMLCa3), 2-63
- User local control B registers
 - (UPMLCb0–UPMLCb3), 2-64
- User performance monitor counter registers
 - (UPMC0–UPMC3), 2-65
- User instruction set architecture (UISA) description,
 - xxi
- USPRG0, 2-31

V

- Vector alignment exception, 10-11
- Vector floating-point data exception, 5-31
- Vector floating-point round exception, 10-10

W

- Watchdog timer interrupt, 5-26
- WIMGE
 - cache-inhibited loads and stores, 12-13
 - coherancy paradoxes, 11-9
 - G bit, 13-14

- general, 12-12
- misaligned accesses and the endian (E) bit, 12-13
- speculative accesses to guarded memory, 12-13
- write-back stores, 12-12
- write-through stores, 12-13
- Write access permission, 11-29
- Write-back
 - definition, 4-4
 - general, 4-10
- Writing to TLBn array, 13-17

X

- XER, 2-3, 2-9